Discovering Large Dense Subgraphs in Massive Graphs

David Gibson Ravi Kumar Andrew Tomkins

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{davgib@us, ravi@almaden, tomkins@almaden}.ibm.com

Abstract

We present a new algorithm for finding large, dense subgraphs in massive graphs. Our algorithm is based on a recursive application of fingerprinting via shingles, and is extremely efficient, capable of handling graphs with tens of billions of edges on a single machine with modest resources.

We apply our algorithm to characterize the large, dense subgraphs of a graph showing connections between hosts on the World Wide Web; this graph contains over 50M hosts and 11B edges, gathered from 2.1B web pages. We measure the distribution of these dense subgraphs and their evolution over time. We show that more than half of these hosts participate in some dense subgraph found by the analysis. There are several hundred giant dense subgraphs of at least ten thousand hosts; two thousand dense subgraphs at least a thousand hosts; and almost 64K dense subgraphs of at least a hundred hosts.

Upon examination, many of the dense subgraphs output by our algorithm are *link spam*, i.e., websites that attempt to manipulate search engine rankings through aggressive interlinking to simulate popular content. We therefore propose dense subgraph extraction as a useful primitive for spam detection, and discuss its incorporation into the workflow of web search engines.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1 Introduction

A canonical challenging problem in the analysis of large graphs is extraction of dense subgraphs. Many graphs, such as the World Wide Web, the telephone call graph, and the global social network graph have different behaviors locally and globally. Globally, these graphs are typically sparse, with a constant average degree. Locally, however, there may be regions that contain far more than their fair share of links: on the web, these regions might be communities or even link spam; in a social network, they might be groups of friends or to a lesser extent geographic localities.

Thus, dense subgraphs often represent cohesive groupings of nodes that are the natural focal points for studying network structure, dynamics, and evolution. Dense subgraph extraction is therefore a key primitive for any in-depth study of the nature of a large graph. Currently, there are some tools that in practice extract small dense subgraphs of a particular form [24, 23, 15, 22, 21], but at heart these algorithms typically look for very small structures containing on the order of ten nodes. There are no efficient techniques in common usage for discovery of large dense subgraphs. Yet real-world graphs do contain locally dense subgraphs whose size is several orders of magnitude larger than what is handled by existing techniques; for example, we show collections of tens of thousands of web sites that link together densely. Thus, the size of locally dense regions may be highly variable, and much of this dynamic range is inaccessible using current techniques.

We propose a new algorithm for this problem based on recursive stages of fingerprinting the graph, which over multiple applications turn dense subgraphs of arbitrary size into fingerprints of constant size. This algorithm is extremely efficient and realizable in the data stream model augmented with sorting and is capable of handling graphs with tens of billions of edges on a single machine with modest resources.

We apply our algorithm to characterize the large, dense subgraphs of a graph showing connections between hosts on the World Wide Web; this graph contains over 50M hosts and 11B edges, gathered from 2.1B pages. We measure the distribution of these dense subgraphs and their evolution over time. We show that more than half of these hosts participate in some dense subgraph found by the analysis. There are several hundred giant dense subgraphs of at least ten thousand hosts; two thousand dense subgraphs at least a thousand hosts; and almost 64K dense subgraphs of at least a hundred hosts.

Upon examination, many of the dense subgraphs output by our algorithm are *link spam*, i.e., websites that attempt to manipulate search engine rankings through aggressive interlinking to simulate popular content. We therefore propose dense subgraph extraction as a useful primitive for spam detection, and discuss how it may be incorporated into the workflow of a standard web search engine with minimal overhead.

The remainder of the paper proceeds as follows. In Section 2, we introduce the basic notation and the background material. In Section 3, we present our algorithm for discovering large and dense subgraphs in massive graphs. This algorithm is based on a recursive application of the shingling algorithm. In Section 4, we discuss the results of running this algorithm on a 50M node graph. In Section 5 we discuss the possibility of using our algorithm to detect link spam. Section 6 contains concluding remarks.

2 Preliminaries

In this section we first provide some background material on graphs, communities, and approximate fingerprinting.

2.1 Definitions and notation

A *directed graph* $G = (V, E)$ consists of a set V of *nodes* and a set E of *edges*, where each edge is an ordered pair of nodes. The *indegree* of a node u is the number of nodes v such that $(v, u) \in E$; here, v is an *inlink* of u. The *outdegree* of u is the number of nodes v such that $(u, v) \in E$; v is an *outlink* of u and the set of outlinks of u is denoted $\Gamma(u)$. There is a *directed path* from u to v in G if there is a sequence of nodes $u = w_1, \ldots, w_k = v$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < k$. An *undirected graph* $G = (V, E)$ consists of a set V of nodes and a set E of edges, where each edge is an unordered pair of nodes; an undirected path between pairs of nodes is defined in an obvious manner. A *connected component* in an undirected graph is a subset of nodes such that for every pair of nodes in the subset, there is an undirected path between the pair.

2.2 Communities in graphs

A *bipartite clique* is a pair of subset $A, B \subseteq V$ of nodes such that $(a, b) \in E$ for every $a \in A$ and $b \in B$. Informally, a *dense bipartite subgraph* is a pair of subset $A, B \subseteq V$ of nodes such that $(a, b) \in E$ for 'most' $a \in A, b \in B$; here, 'most' parametrizes the density of the subgraph. A *clique* in an undirected graph is a subset $A \subseteq V$ of nodes such that $(a, b) \in E$ for every $a, b \in A$. Unfortunately, finding bipartite cliques and dense subgraphs are notoriously hard combinatorial problems, even to solve approximately (see, e.g., [13]). We therefore resort to heuristics that are simple, efficiently implementable, and effective.

Early work studied *online communities* in the context of hypertext and content analysis [7, 25]; web communities were first studied in [24, 23, 15]. Kumar et al. [24] defined communities to be dense bipartite subgraphs. Their hypothesis was that any topically focused community on the web is likely to contain a dense bipartite subgraph (the *signature*) and almost every occurrence of the signature corresponds to a web community. Their algorithm was a two-step process—a careful enumeration and removal of small-sized bipartite cliques, followed by an *apriori*-style [3] enumeration algorithm on the residual, hopefully smaller, graph. Abello et al. [1] consider dense subgraph extraction based on a notion of pruning candidate sets of nodes, in a model in which the nodes but not the edges of the graph may be memory-resident. Flake et al. [15] adopted a more sophisticated definition of a web community based on network flow and Kumar et al. [21] use a local search heuristic to find dense bipartite communities with certain special properties. In presenting our algorithm for finding large dense subgraphs, we will discuss the differences in the problem domain that make existing algorithms ineffective.

2.3 The shingling algorithm

Shingles were introduced in [9], and have since seen wide usage to estimate the similarity of web pages using a particular feature extraction scheme based on overlapping windows of terms (motivating the name "shingles"); they were also used to detect mirror sites (see Chakrabarti [11] and Bharat et al. [5]). We will employ the technique to solve the following problem: given a subset S of a universe U of elements, generate a constant-size fingerprint such that two subsets A and B may be compared by simply comparing their fingerprints.

A simple and natural measure of the similarity of two sets is the *Jaccard coefficient*, defined as the size of the intersection of the sets divided by the size of their union: $|A \cap B|/|A \cup B|$. Formally, if π is a random permutation of the elements in the ordered universe U from which A and B are drawn, then it can be shown that (see, e.g., [8])

$$\Pr[\pi^{-1}(\min_{a \in A}\{\pi(a)\}) = \pi^{-1}(\min_{b \in B}\{\pi(b)\})] = \frac{|A \cap B|}{|A \cup B|}.$$

That is, the probability that the smallest element of A and B is the same, where smallest is defined by the permutation π, is exactly the similarity of the two sets according to the Jaccard coefficient. Using this observation, we compute the fingerprint of A by fixing a constant number c of permutations $\pi_1, \ldots, \pi_c$ of U, and producing a vector whose i-th element is $\min_{a \in A} \pi_i(a)$. The similarity of two sets is then estimated to be the number of positions of their respective fingerprint vectors that agree.

This formulation is not yet sufficient for our needs; we require one generalization. The formulation as given may be viewed as follows: consider every one-element set contained entirely in A or B, and measure agreement by the fraction of these one-element subsets that appear in both sets. Generalizing, we may instead consider every s-element set contained entirely within either set, and measure similarity by the fraction of these s-element subsets that appear in both. This is identical to measuring the similarity of A and B by computing the Jaccard coefficient of two sets A_s and B_s, where $A_s = \{\{a_1, a_2, \ldots, a_s\} \mid a_i \in A\}$, and B_s is defined likewise. The same fingerprinting scheme applies unchanged to A_s and B_s. We will refer to each of the s-element subsets as a *shingle*, and to the algorithm that produces the set as an (s, c) shingling algorithm. By tuning both s and c, we may arrive at an algorithm that accurately distinguishes between sets that are above or below a certain threshold of similarity; see Section 3.5 for a discussion.

Fortunately, we need not consider all possible permutations π in our choice of c such permutations; a more succinct family of *min-wise independent permutations* will accomplish the task. For more details, see [9, 8]. In practice, we employ two-universal hash functions, which have been shown to work well.

In the first step of our algorithm, each set will represent the outlinks of a particular node, and two nodes will be considered similar if they share many outlinks. In later stages of the algorithm, we will employ exactly the same formulation recursively to determine the similarity between two shingles in terms of the set of nodes that contain each shingle. A time and memory efficient (s, c) shingling algorithm for a set $\{a_1, \ldots, a_n\}$ is as follows:

```
**Algorithm Shingle**($a_1, \ldots, a_n, s, c$)
    Let $H$ be a hash function from strings to integers
    Let $p$ be a large random prime (say, 32 bits)
    Let $a_1, b_1, \ldots, a_c, b_c$ be random integers in $[1 \ldots p]$
    For $i = 1$ to $n$ do $x_i = H(``a_i")$
    For $j = 1$ to $c$ do
        For $i = 1$ to $n$ do $y_i = (a_j * x_i + b_j) \bmod p$
        Let $y'_1, \ldots, y'_s$ be $s$ minimal elements of $y$
        Let $z_j = H(``y'_1 \circ \cdots \circ y'_s")$
    Output $z_1, \ldots, z_c$
```

3 Algorithm for discovering large dense subgraphs

In this section we present our algorithm for extracting large dense bipartite subgraphs in massive graphs. The goal of the algorithm is to enumerate as many such disjoint subgraphs as possible. As we mentioned earlier, this problem has been considered before [24, 23, 15, 22, 21], so we would prefer to apply known techniques to our case. Unfortunately, we are unable to do so for the following reasons. Existing algorithms for identifying dense subgraphs perform well when the size of the subgraph is manageably small. The pruning and *a priori*-based methods in [24, 23], the network flow approach of [15], and the local search heuristic of [21] are effective only so long as the size of the dense subgraph is small, say, a few tens of nodes. But our goal is to find dense subgraphs that are large: say, of the order of tens of thousands of nodes. This, plus the possibility that there could be thousands of such subgraphs, necessitates new methods for finding dense subgraphs. Furthermore, our goal is to find as many of the dense subgraphs as possible or at least find most of the nodes in these dense subgraphs.

The desirable features of a good algorithm for finding dense subgraphs are that it should:

1. be able to identify *large* dense subgraphs and in the worst, be able to identify most of the nodes in large dense subgraphs,
2. be extremely *efficient* in terms of running time,
3. preferably be realizable in the *data stream* model [4, 20], i.e., use very little main memory and process data on the fly with read/writes to secondary storage, and
4. be *scalable.*

Below, we describe an algorithm that fulfills these desiderata.

3.1 A high-level description

Our algorithm seeks clusters of nodes that tend to link to the same destinations, and proceeds as follows. First, for each node in the graph, it applies an (s, c) shingling algorithm to the set of destinations linked-to from that node, resulting in c shingles per node. For each distinct shingle created by this process, it produces a list of all nodes containing that shingle. We have now succeeded in bringing together the nodes that share a particular shingle (and therefore a particular set of s outlinks). In order to begin clustering nodes together, we must find sets of nodes that jointly share a sufficiently large number of shingles. In the next phase, we would like to perform one further level of analysis, grouping together shingles that tend to occur on the same nodes, so that we may use these sets of commonly co-occurring shingles as the defining patterns of a dense subgraph. Such an analysis is yet another application of the (s, c) shingling algorithm, this time to the set of nodes associated with a particular shingle, and results in bringing together shingles that have significant overlap in the nodes on which they occur. If necessary, the sequence of operations may be repeated for as many levels as required, as shown in Figure 1. In practice, we will now show that two levels of this algorithm suffice to convert dense subgraphs of arbitrary size (i.e., hundreds or hundreds of millions of nodes) into small-size fingerprints that can then be recognized in a straightforward manner. The particular density threshold captured by the scheme is determined by the parameters s and c.

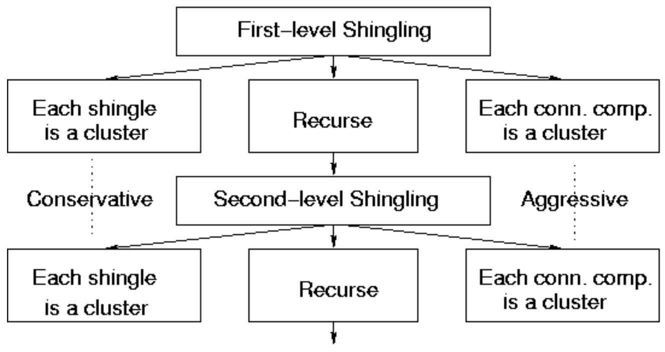
First–level Shingling
Each shingle is a cluster
Recurse
Each conn. comp. is a cluster
Conservative
Second–level Shingling
Aggressive
Each shingle is a cluster
Recurse
Each conn. comp. is a cluster

Figure 1: Recursive shingling flowchart.

3.2 An example: complete subgraphs

For an extreme example of a dense subgraph, consider a subgraph consisting of n nodes, each of which has an edge to every one of the n nodes, i.e., an n-clique with self-loops. We will walk through the stages of the algorithm on this subgraph, as represented pictorially in Figure 2. The discussion below covers nodes v_i that might have an outlink to a node O_j. While all of these are nodes in the graph, we will adopt the notational convention that v refers to a node that is being clustered together with other nodes based on its outlinks, while O refers to an outlink.

Initially, the subgraph consists of n nodes $v_1, \ldots, v_n$, each of which contains n outlinks $O_1, \ldots, O_n$. At this point, the representation of the subgraph requires quadratic space in n. We apply our (s, c) shingling algorithm to the outlinks of each node, creating c shingles of size s. For concreteness, assume $s = 3$. Thus, the first shingle for node v_i might be $S_{i,1} = (O_4, O_7, O_{22})$. If a node had slightly different outlinks, then many of its shingles would be identical to the shingles of other nodes, but a few of its shingles might differ. To complete the first-level shingling, we reformat the file to list each distinct shingle along with all the nodes containing this shingle; we describe below how to accomplish this reformatting efficiently.

Original: $n \times n$ size

$$\begin{array}{rcccc} v_1: & O_1, & O_2, & \cdots & O_n \\ v_2: & O_1, & O_2, & \cdots & O_n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ v_n: & O_1, & O_2, & \cdots & O_n \end{array}$$

After first-level shingling: $c \times n$ size

$$\begin{array}{rcccc} S_1: & v_1, & v_2, & \cdots & v_n \\ S_2: & v_1, & v_2, & \cdots & v_n \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ S_c: & v_1, & v_2, & \cdots & v_n \end{array}$$

After second-level shingling: $c \times c$ size

$$\begin{array}{rcccc} M_1: & S_1, & S_2, & \cdots & S_c \\ M_2: & S_1, & S_2, & \cdots & S_c \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ M_c: & S_1, & S_2, & \cdots & S_c \end{array}$$

Figure 2: Reduction of data by recursive shingling.

In our clique example, each node contains the exact same outlinks and thus contains the exact same shingles, so each shingle occurs on all n nodes; we therefore represent a shingle as simply S_j. This results in the representation in the middle of the figure, which has moved from an $O(n^2)$ size layout to an $O(c \cdot n)$ size layout.

The first-level shingles could in some extremal cases be used to detect dense subgraphs without any further work. The most conservative approach to doing so is to cluster all nodes that have identical first-level shingles. In our example of a complete subgraph, this will successfully return the clique as a cluster. However, as soon as the clique becomes simply a dense subgraph, the nodes begin to differ slightly in their shingles, and the output begins to fragment. On the opposite end of the spectrum is the most aggressive scheme possible given the available data: begin with every node in a separate cluster, and repeatedly merge pairs of clusters A and B as long as the same shingle appears in some node of A and some node of B. This scheme may introduce spurious clusters because many sites share a few popular outlinks, such as `microsoft.com` or `adobe.com`, and consequently share a shingle.

Rather than adopting either of these extreme schemes, a natural approach is to follow the middle ground and declare two nodes to belong to a dense subgraph if and only if they have significant overlap in their shingles. Fortunately, a recursive application of our shingling algorithm may be employed to exactly this purpose; this is the key idea behind our technique. We apply it in the first level to find groups of nodes that have significant overlap in their outlinks; we may apply it again to find groups of shingles that have significant overlap in the nodes that contain them.

Back to our example of the clique, we now pick for each distinct first-level shingle S_i a set of c "fingerprints," where each fingerprint now consists of a small set of s nodes which contain S_i. For example, shingle S_1's first fingerprint might be (H_7, H_8, H_{20}). This fingerprint is a *second-level shingle*, or a *meta-shingle*, which we will refer to as M_j. We then reformat as detailed below to find the set of all shingles that share some meta-shingle; that is, the set of all shingles which appear jointly on a particular fingerprint set of nodes, resulting in the representation at the bottom of Figure 2. But observe the data reduction which has occurred as part of this operation. By applying the first-level shingling, we moved from the initial $O(n^2)$ representation to a new representation of size $O(c \cdot n)$. In applying the second-level shingling, we again improved our representation size down to $O(c^2)$. So for any graph, any clique of any size will be converted into a set of shingles, whose size is independent of the size of the clique. The process behaves similarly for subgraphs above a certain density (dependent on the count and size of shingles being produced at each level), giving us small representations of large dense subgraphs in the original graph.

3.3 The algorithm

We describe the two main components of the algorithm—the recursive shingling (Section 3.3.1) and the clustering that we apply once the last round of shingling terminates (Section 3.3.2).

3.3.1 The recursive shingling step

We now formally describe the two-level recursive shingling algorithm described in the example above. This algorithm applies an (s_1, c_1) shingling algorithm first to the outlinks of each node and then applies an (s_2, c_2) shingling algorithm to the first-level shingles. The algorithm proceeds as follows.

```
**Algorithm Shingle2**$(v_1, \ldots, v_n, s_1, c_1, s_2, c_2)$
    For $i = 1$ to $n$ do
        $S_1(v_i) = \textbf{Shingle}(\Gamma(v_i), c_1, s_1)$
    Let $S = \cup_{i=1}^{n} S_1(v_i)$
    For $s \in S$ do
        Let $\Gamma(s) = \{v \mid S_1(v) \ni s\}$
        $S_2(s) = \textbf{Shingle}(\Gamma(s), c_2, s_2)$
    Let $T = \cup_{s \in S} S_2(s)$
    For $t \in T$ do
        Let $\Gamma(t) = \{s \in S \mid S_2(s) \ni t\}$
        Output $\langle t, \Gamma(t) \rangle$
```

The above algorithm can easily be realized in the data stream model, together with a sorting primitive, as described below. As in the algorithm, let $\Gamma(v)$ be the set of outlinks of node v. Assume that the input is presented in the form of $\langle v_i, \Gamma(v_i) \rangle$ for every node v_i, i.e., each node appears together with its outlinks (the adjacency list representation). As the given graph is read in this form, the c first-level shingles of node v_i may be written out on the disk as c distinct lines of the form $\langle s_j, v \rangle$ for each shingle $s_1, \ldots, s_c$.

These pairs may then be sorted by the first column using an external sorting algorithm, and then aggregated to produce a file of the form $\langle s, \Gamma(s) \rangle$ containing for each shingle s the nodes that contain that shingle. This file has the same format as the original file, so the same code may be applied to perform the second-level shingling. The output of the second-level shingling is once again sorted and aggregated to be of the form $\langle t, \Gamma(t) \rangle$ where t is a second-level shingle and $\Gamma(t)$ are all the first-level shingles that share a second-level shingle.

3.3.2 The clustering step

Finally, we must define a notion of clusters of first-level shingles. Let S be the set of all first-level shingles; that is, shingles produced by the application of the (s_1, c_1) shingling algorithm to the set of outlinks of a node. Each such shingle corresponds to a set of s_1 outlinks. Then let T be the set of second-level shingles; that is, the shingles produced by application of the (s_2, c_2) shingling algorithm to the set of nodes corresponding to a first-level shingle. Each such shingle corresponds to a set of s_2 nodes. We say that two first-level shingles $a, b \in S$ are related if and only if they share a second-level shingle. Clusters of first-level shingles are then equivalence classes under this relation. Algorithmically, we create an undirected graph G_S whose nodes represent the first-level shingles and whose edges represent the above relation. Clusters of first-level shingles then correspond to connected components in G_S.

To efficiently find connected components in this graph, we resort to the classical union-find algorithm. This algorithm maintains a family of sets via a standard union-find data structure. Initially, the family contains only singleton sets, each consisting of exactly one node of the graph. For every edge (u, v), the sets in the family containing u and v are merged. It is easy to see that after all the edges are processed, each set in the family contains the nodes of a connected component. In our case, notice that the output of **Shingle2** is of a special form—$\langle t, \Gamma(t) \rangle$, which means that G_S is a union of cliques, i.e., G_S has edges of the form (s_1, s_2) for every $s_1, s_2 \in \Gamma(t)$. Therefore, the connected components algorithm can exploit this property and work with an *implicit* representation of the edges of G_S rather than constructing them explicitly; this represents a massive savings in terms of computational time and space. Note that the only place where main memory is used is to store the sets in the family—this can be done in space linear in the number of nodes in G_S. The rest of the processing can be done in the data stream model. To reduce space requirements further, we may discard second-level shingles t's with $|\Gamma(t)|$ very small. This compromises the correctness marginally, but as we argued earlier, nodes in a large dense subgraph will share several second-level shingles, and therefore this step will not have any drastic effects in terms of missing such subgraphs. The algorithm is described below.

```
Algorithm CC $(t_1, \Gamma(t_1), \ldots, t_m, \Gamma(t_m))$
  Let $V_S = \cup_{i=1}^{m} \Gamma(t_i)$
  Let $\mathcal{C} = \{\{s\} \mid s \in V_S\}$
  For $i = 1$ to $m$ do
      Merge the sets $\{$ Find$(s) \mid s \in \Gamma(t_i)\}$ in $\mathcal{C}$
  For $C \in \mathcal{C}$
      Output cluster $\{s \mid s \in C\}$
```

Furthermore, in the case of graphs so large that not even the list of nodes will fit in main memory, this algorithm may be replaced by a randomized algorithm in the data stream model augmented with an external sort primitive using just logarithmic memory and a logarithmic number of passes; for details of this data stream algorithm, see [2]. Thus, finding connected components is not a bottleneck even if the number of nodes far exceeds the size of main memory.

3.3.3 Putting it all together

We now present the entire algorithm for detecting dense subgraphs. Given a graph, we first run the two-level shingling algorithm **Shingle2** and then identify clusters of first-level shingles using the connected-components algorithm **CC**. At the end of this step, we have sets of first-level shingles that need to be mapped back to the nodes of the given graph. To do this, we merge the clusters $\mathcal{C}$ of first-level shingles and the output S of the first-level shingle step; this can be accomplished by sorting on the first-level shingle. The algorithm is described below.

```
Algorithm DenseSubgraph $\langle v, \Gamma(v)\rangle$
  Choose $c_1, s_1, c_2, s_2$
  Shingle2$(\langle v, \Gamma(v)\rangle, c_1, s_1, c_2, s_2)$
  Let $S = \langle s, \Gamma(s)\rangle$ be first-level shingles
  Let $\langle t, \Gamma(t)\rangle$ be second-level shingles
  $\mathcal{C} =$ CC$(\langle t, \Gamma(t)\rangle)$
  For $C \in \mathcal{C}$ do
      Output $\cup_{s \in C}\Gamma(s)$ as a dense subgraph
```

3.4 Highlights of the algorithm

Our new algorithm has many advantages, especially scalability and efficiency.

(1) As we illustrated throughout the description of the algorithm, the entire algorithm has a very efficient implementation in the data stream model using basic primitives such as sorting and merging. The number of passes made over the input and the amount of main memory used is very small, enabling the algorithm to handle graphs with tens of billions of edges even on machines with modest amount of main memory (as we demonstrate in the experiment).

(2) As we argued in 3.2, the number of second-level shingles is *independent* of the size of the dense subgraph, especially if its density is high.

(3) As in Figure 1, our algorithm can be extended to hierarchically decomposing the given graph into dense structures in the following way. For each of the first-level shingle clusters, the shingling algorithm can be applied once more with stringent parameters s and c. In this manner, a hierarchical decomposition can be obtained.

(4) The algorithm works to identify both bipartite and directed cliques and can be trivially extended to work for undirected graphs as well.

3.5 Discussion

We remark briefly on the theoretical behavior of the recursive shingling algorithm. For simplicity, consider an Erdös–Renyí random graph $G_{n,p}$. We shall examine how the first-level shingle performs on this graph for various values of p. The function

$$h(s, c, p) = 1 - (1 - p^s)^c$$

captures the probability that at least one shingle in an (s, c) shingle is shared when the outlinks of a node are shingled; this corresponds to the correctness of the procedure. Figure 3 shows the effect of this correctness for various values of c and s. Consider two nodes whose outlinks have a Jaccard similarity coefficient of p. Looking up p on the x axis of the figure shows for various different curves representing settings of c and s the probability that the two nodes will share at least one shingle, a critical enabler of our algorithm.

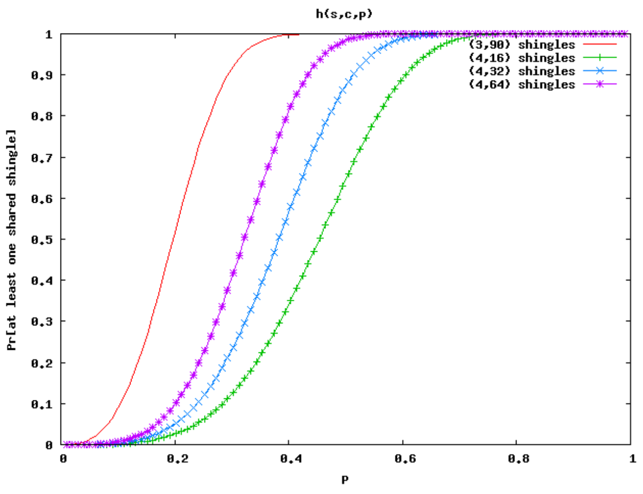

Figure 3: Effect of c, s on the correctness of shingles.

4 Dense subgraphs in the host graph

In this section we apply the algorithm of Section 3 to discover large collections of densely-interlinked websites on the World Wide Web. We study the *host graph* (e.g, [6]), a directed graph in which nodes are hosts and edges are present from a source host to a destination host whenever a page of the source host links to a page on the destination host. More formally, host t (the destination) is an outlink of host s (the source), i.e., $t \in \Gamma(s)$, if some page hosted on host s contains a hyperlink to some page hosted on t. Variants of this model could also incorporate the multiplicity of links from s to t but we do not consider these in this paper.

4.1 Data set

IBM's WebFountain project [18, 12] performs large-scale analysis of unstructured information, including the web. We make use of the WebFountain crawl, which contains over 2B pages and 50M sites. Throughout this analysis, we use the word *site* to mean all the pages that are retrieved from a particular hostname.[1]

WebFountain includes a *site store* designed to aggregate information across all the pages of a site. It is rebuilt at regular intervals, using the most recent data from the crawler to update and replace the set of pages which it covers. It consists of a record for each site which is completely precomputed so that all the data is available instantaneously at query time. The results of each build have been archived over the period of June to October 2004, so that we can monitor the changes that have occurred in web content and structure over this interval. The site store grew from 40 million sites in June 2004 to 55 million in October. This growth reflects both natural growth in the web and increased coverage of our crawler. Further details on the architecture of the site store are discussed in [17, 16].

Note that the update process does not currently report 404s or any other form of site and page disappearance to the site store. Thus the data set that we have gathered documents the graph as of the last time each page was successfully crawled: new copies of pages replace their older version, but pages that have disappeared still contribute edges to the graph.

For our experiments, we focus on the link data in the site store: for each site we keep all the other sites referenced, along with a count of the number of links to each site. This forms a labeled directed host graph. This host graph consists of approximately 50M nodes, representing about 2.1B underlying web pages. There are around 11B edges, implying a mean outdegree of around 220.

[1]This definition of site is not completely accurate: often sites run by large entities will serve a coherent set of content from several hostnames, and the same hostname may also serve content from several separate and independent entities. The single hostname per site rule, however, allows us to avoid the error-prone detection of such cases and works correctly for a large majority of sites.

4.2 Results

The first operation we performed was a two level $(4, 16)$ shingling, whose goal was to pull together hosts with a relatively large fraction of outhost overlap.

The first-level shingling resulted in 17G of output (5.5G compressed) containing 275M distinct first-level shingles, and 420M host occurrences. The number of hosts per shingle averaged between one and two, but included almost one thousand giant shingles containing in excess of ten thousand hosts.

The second-level shingling of this file produced a 1.7G result (690M compressed) containing 60M distinct second-level shingles, and 98M total occurrences of first-level shingles. As described in Figure 2, a clique of size n produces c shingles of size n at the first level, but only c shingles of size c at the second level. Large shingles at the second level are instead the result of large semi-dense graphs with random linking patterns. In fact, the 60M second-level shingles contain no shingle that corresponds to more than one hundred first-level shingles, and very few shingles whose size is larger than $c = 16$. For each shingle size between five and sixteen, we see between 200K and 400K such shingles. Once we reach size 17, we see fewer than 3000 such shingles, and fewer than 1500 shingles that are larger than 17. Thus, the second-level shingling has done an effective job of producing an essentially constant-degree graph for processing.

At this point, the connected components algorithm produced 2.8M components whose number of first-level shingles is shown in the histogram of Figure 4.

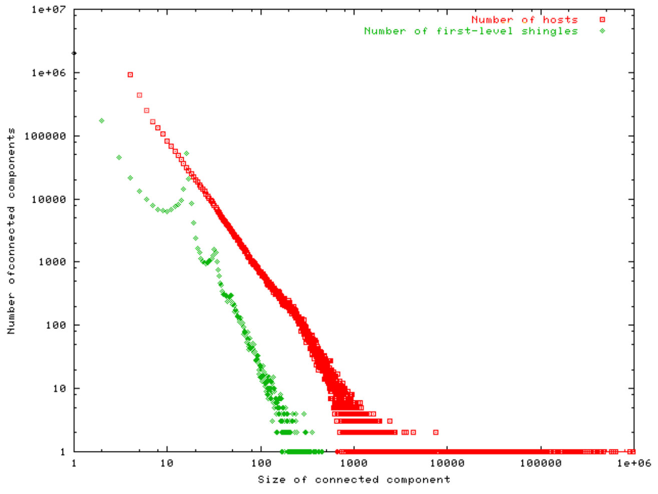

Figure 4: Histogram of cluster sizes in both hosts and first-level shingles, after connected components completes on second-level $(4, 16)$ shingles.

Expanding first-level shingles back to their hosts resulted in (an average of) 2.5 shingles getting mapped to (an average of) 24 hosts. The component sizes are shown overlaid in Figure 4.

We turned next to a more aggressive $(3, 90)$ shingling algorithm; this algorithm runs slower than the $(4, 16)$ shingling algorithm. As Figure 3 shows, these values provide a tighter separation and capture hosts with a lower fraction of overlap, down to about 25% with 95% accuracy. The first-level shingling resulted in 100G of output (30G compressed) containing 957M distinct shingles and 2.5B host occurrences. All subsequent recursive steps were performed using a $(4, 16)$ shingling algorithm.

The second-level shingling resulted in 24G (8.7G compressed) of output containing 1B distinct second-level shingles, and 1.2B total occurrences of first-level shingles. Connected components on this set resulted in a giant component of 1.8M shingles, showing insufficient convergence at the second level.

We performed a third-level shingling, which resulted in 16G of data (5.8G compressed), 700M distinct third-level shingles, and 750M occurrences of second-level shingles. Let the "size" of a shingle during a particular operation be the number of occurrences of that shingle, so the size of a first-level shingle is the number of hosts for which the shingle is present, and so forth for higher-level shingles. Figure 5 shows histograms reporting how many shingles of each size appear in the first-, second-, and third-level shinglings of the $(3, 90)$ shingle dataset. As the figure shows, the first-level shingling contains many enormous shingles, some in excess of 100K hosts. However, a key property of recursive shingling now comes into play. The second-level shingles all have size less than 100, although there are a great many of them. There is nonetheless a giant component in the resulting constant-degree graph. The third-level shingles all have size at most 10, and the connected components computation on this graph terminated with 110M of compressed output and does not contain a giant component. The resulting components of second-level shingles were mapped back to first-level shingles, and then to hosts. The histograms for the number of hosts in each component of the final output of the second-level and third-level connected component operations are shown in Figure 6. The sizes of host clusters for the third-level shingles show a peak at around 42 hosts. This peak comes about due to the filtering of small-outdegree shingles described in Section 3.3.1. We filtered first-level shingles with outdegree less than six. Thus, the outdegree of each first-level shingle is manually restricted to be at least six. The second-level shingles are processed using a $(4, 16)$ shingling, and so must have outdegree at least four to generate a third-level shingle. Thus, even a component of size 1 will map back to four second-level shingles, and 24 first-level shingles. Unless there is duplication among the corresponding hosts, clusters of size less than 24 will therefore not be found by these settings. Nonetheless, this filtering for efficiency does not impact the discovery of larger communities of fifty or more hosts, and these larger clusters result in the smooth tail of the curve in the figure.

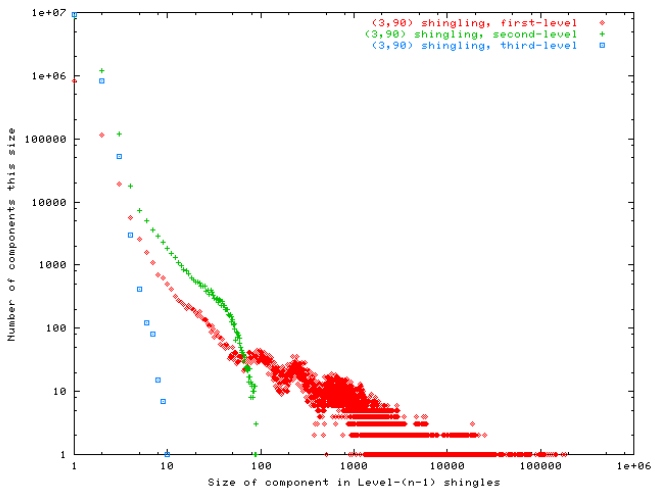

Figure 5: Histogram of cluster sizes in first, second, and third levels for $(3, 90)$ shingles.

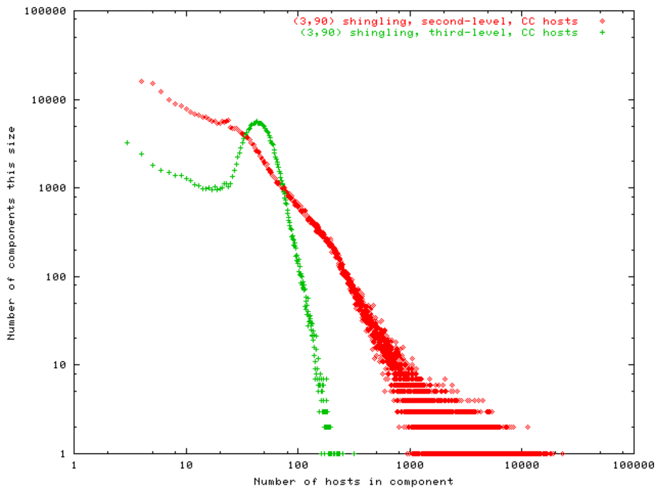

Figure 6: Histogram of cluster sizes in second and third level connected components for the $(3, 90)$ shingles.

4.3 Correctness of extracted subgraphs

We considered a random sample of one hundred dense subgraphs output from the algorithm, and manually evaluated whether the dense interlinking was spurious (as, for example, if several sites linked to Adobe, Microsoft, and Google). Our conclusion is that, after the second-level shingling, every subgraph output by the algorithm represents a set of “truly” interlinked sites, according to human judgment.

4.4 Temporal evolution of dense subgraphs

In this section, we perform a first-order evaluation into the growth patterns of dense subgraphs, and show that, in aggregate, growth is more variable than the growth of individual web sites, supporting the intuition that growth in the graph is likely to be focused on particular subgraphs; for example, subgraphs representing communities that are “hot.”

We begin with a baseline study of the growth of individual sites. While the outdegree of a site is easy to measure, it is under the control of a single entity and does not accurately track the popularity of a site in the graph at large. Instead, we study the indegree of a site, which is an emergent property of the graph, and is known to be a more robust feature (see, e.g., [10]).

We generate samples of several thousand sites divided into the following seven categories: first, hosts where there were no inlinks in June, but some in October; second, hosts where the number of inlinks was 5 to 9 in June; third, hosts where the number of inlinks grew more than tenfold over the interval; and fourth through seventh, hosts where the number of inlinks was exactly 10, 100, 1000, and 10000 in June.

Figure 7 shows the number of inlinks to 1400 sites drawn randomly from these seven categories, from the period from 6/2004 to 9/2004. A typical site grows by a factor of about 1.1 in the interval from June to September. The number of inlinks in this plot are taken to be the number of unique inlinking IP addresses, in order to minimize artifacts in the growth measurement due to collections of spam sites hosted from the same address.

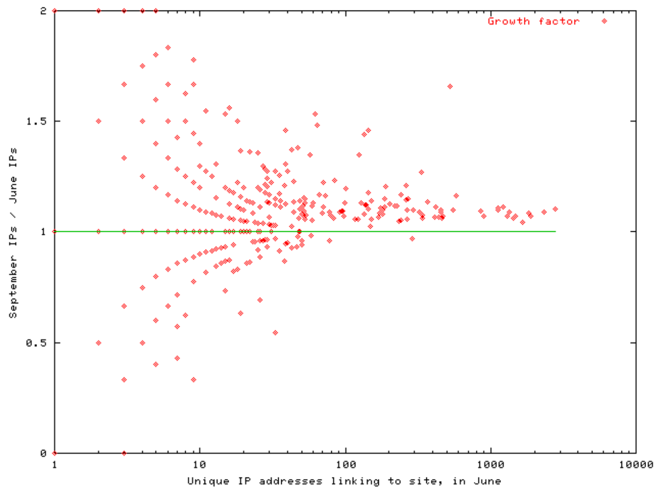

Figure 7: Growth rates of number of inlinking IPs to 1400 sites chosen from a distribution of various site types.

The inlink growth curves are surprisingly smooth and consistent. As this sample is broad in its coverage, two conclusions may be drawn from it: many sites do not grow at all and the initial growth of sites is very small.

Using this information as a baseline, we proceed to analyze the dense subgraphs produced above. We produced a random sample of one hundred dense subgraphs. Each such subgraph is characterized by a set of nodes which share a significant fraction of outlinks. We extracted the *center-set* of those subgraphs; specifically, the set of up to eight sites that have the largest number of inlinks within the subgraph, excluding sites that are linked-to from more than one subgraph. We now choose a random sample of four hundred such centers in order to evaluate their growth. The results are shown in Figure 8. The plot shows the rate of growth over time of the number of hosts that link to any node of the center-set. Each short line segment plots the number of inlinks over the time interval from June to October. The line segments are arranged in arbitrary positions along the x axis, and colored arbitrarily, in order to convey four hundred distinct growth patterns in a single image; the x axis thus does not represent absolute time.

The curves show dramatically that, while many sites display little variation in the y axis, there are also many sites that grow dramatically during the three months of our study. In fact, the curves that show significant motion in the y axis typically do so in one or sometimes two time periods. An examination of the sites themselves shows that they are typically rapidly growing spam link farms.

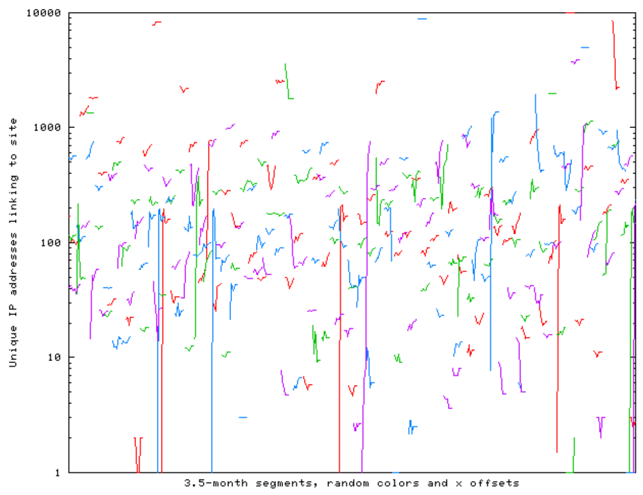

Figure 8: Growth of center-sets over eight timesteps.

Figure 9 summarizes the rates of growth and shrinkage in the above plot, overlaid against the growth and shrinkage rates for individual web sites. The left-hand group of ten data points in the plot represents sites or center-sets that have shrunk; each point is a decile, with the rightmost of the ten points representing shrinking by 10% while the leftmost represents shrinking by 100%. The right-hand points represent growth, with the first point representing growth of 10% and the rightmost point representing growth of 100% or more. The sites chosen in the site growth curve are chosen uniformly from our sample sites whose size in June is 10, 100, 1k, and 10k pages. Each of the four categories taken individually shows a similar curve; we have coalesced them for clarity of presentation. Overall, center-sets are seen to show significantly more variation in growth and shrinkage than random sites. Dense subgraphs, which have been known to represent good target areas for mining topical information from the graph (see, e.g., [23]), thus also represent focused areas of change in the graph. This suggests an interesting direction for future work: is it possible to understand the nature of this change with respect to a particular subgraph, and can such an analysis tell us important properties such as whether the subgraph represents link spam?

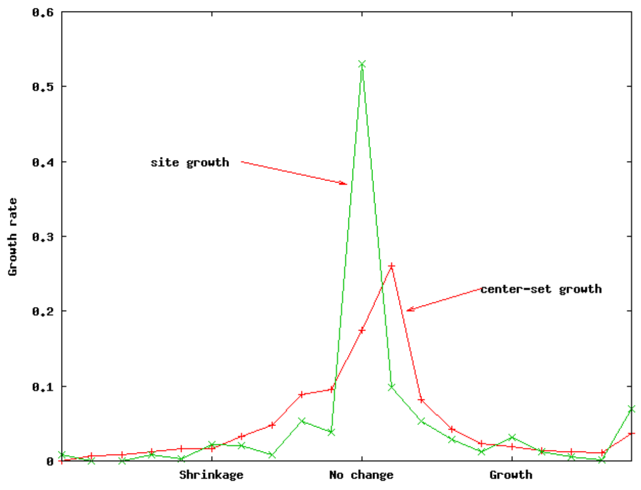

Figure 9: Summary of growth rates for sites versus center-sets.

5 On detecting link spam

Link spam refers to websites that attempt to manipulate search engine rankings through aggressive interlinking to simulate popular content. While measurements are difficult, it is generally believed that a nontrivial fraction of the web is spam of one form or another; the Search Engine Optimization community typically reports a number from 25% to 40%, while Fetterly et al. [14] have reported 5%. These measurements differ radically based on various definitions of spam and data collection methodologies, but it is widely acknowledged that link spam is a large and growing problem. These sites leach valuable bandwidth and computational resources from crawlers, burden indexers with additional storage and computational requirements, and have significant negative impact on the ability of users to find content they seek. It is widely acknowledged (see, e.g., [19]) that link spam is one of the most important challenges to web search engines.

In the course of performing the experiment described above, we were struck by the extent to which the resulting large, dense subgraphs seemed biased towards link spam, compared to the underlying dataset. In order to evaluate this theory, we considered one hundred sites chosen as follows. First, we randomly selected one hundred nodes that participate in a dense subgraph as output by a two-level $(4, 16)$ shingling algorithm; the algorithm has thus detected that each of these nodes shares a significant fraction of outlinks with various other nodes. Next, for each node v drawn from these hundred nodes, we considered every outlink of v, and removed those were also linked-to from a different dense subgraph of the analysis. We ranked the remaining nodes by the number of inlinks from the dense subgraph containing v. We selected the top eight such nodes, and chose one at random for evaluation. We evaluated these 100 sites, and discovered that 88% were link spam. Thus, most of the large dense subgraphs of the host graph are actually link spam, and detection of dense subgraphs represents a potential approach to addressing part of the link spam problem, with the following desirable properties:

- Our algorithm operates at the level of web sites rather than individual pages, matching the granularity of the vast majority of link spam.
- Our algorithm is efficient, and we show below that it may readily be incorporated into the workflow of a standard search engine.
- Approaches based on dense subgraph identification force spammers to spend more time and effort constructing dense subgraphs that more closely mimic organic growth; this additional cost reduces the economic motivation for spam creation.

Once dense subgraphs have been extracted, there are a number of natural directions of future research for making use of them in link spam analysis, based on statistical tests that could be performed to determine whether two sites are jointly operated link spammers. Examples include: Do the sites share a disproportionate number of outlinks? Do the sites update at the same time or add links to the same destination at the same time? Do inlinks to the two sites often arrive at the same time? Do the sites contain identical templates or similar content shingles? Our algorithm address only the first question above: do the two sites share more outlinks than they “should”.

5.1 Incorporating dense subgraph extraction into search engines.

Large, dense subgraphs are of interest to search engines for several reasons, including link spam, but also more positive motivations such as identifying densely-linked high-quality communities which may be employed to answer certain types of queries more effectively. We speak briefly to the architectural changes to implement our subgraph detection algorithm in the architecture of a typical search engine. We will assume in this discussion that a two-level recursive shingling followed by a connected component algorithm will suffice.

Observe that large dense subgraphs may be extracted without aggregating the host graph, but that the local graph corresponding to a single crawl node must be dumped. We assume that the outlinks for each host crawled by a certain node have been dumped. The first step, then, is to perform first-level shingling of the outlinks of each host. This operation requires simply that c heaps are maintained, each containing s elements, and each new element is hashed according to the c different hash functions and inserted into each of the heaps. This requires very little memory. After the site has been scanned in this manner, the outhosts in each heap can be dumped and hashed together to form the c output shingles for the host, and c records of the form $\langle$host, shingle$\rangle$ can be dumped to a large file. This file should now be aggregated across all the nodes of the crawler, and sorted by shingle. The 200-degree host graph need never be shipped, just the c shingle values per host.

Once the sort completes, the hosts corresponding to a certain first-level shingle may be aggregated. The resulting file can now be processed in exactly the same manner, as it consists of a shingle followed by a number of outhosts. In our implementation, the same code is used to perform shingling at all levels, from the host graph onwards. After the second-level shingling has been performed, each line of the resulting file will consist of a second-level shingle followed by a reasonably small number of first-level shingles. The connected component operation can now the run in streaming mode in order to group together all the first-level shingles that share at least one second-level shingle. By sorting these by shingle, and likewise sorting the original dump file by shingle, the hosts can be reintroduced into each component. This completes the operation, and the resulting large dense subgraphs may be processed as necessary. No step of this operation requires more memory than is available on a single reasonable machine.

6 Conclusions

We have presented a new algorithm for detecting large, dense subgraphs in massive graphs. Our algorithm is based on a recursive application of shingling followed by a final clustering step. Since the basic steps can be implemented in the data stream model, our algorithm is extremely efficient and scalable: it can handle graphs with billions of edges, while requiring only modest amounts of main memory. We have applied this algorithm to detect large, dense subgraphs in the host graph of the world wide web, containing 50M nodes and 11B edges, and found dense subgraphs containing order of tens of thousands of hosts. As far as we know, this is the first web-scale graph analysis scheme capable of detecting dense subgraphs of this size with limited resources.

We have analyzed the resulting subgraphs, and determined that many results from link spam. Our algorithms thus generate a useful feature that may be employed in the battle against link spam. We show that our subgraph extraction algorithm may be implemented into the workflow of a web search engine with minimal overhead. Future work includes comparing the performance of our algorithm to others, including the peeling algorithm of [1].

References

[1] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *Proc. 5th Latin American Symposium on Theoretical Informatics*, pages 598–612, 2002.

[2] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proc. 45th IEEE Annual Foundations of Computer Science*, pages 540–549, 2004.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[5] K. Bharat, A. Broder, J. Dean, and M. R. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. *Journal of the American Society for Information Science*, 51(12):1114–1122, 2000.

[6] K. Bharat, B.-W. Chang, M. R. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between web sites. In *Proc. 2001 IEEE International Conference on Data Mining*, pages 51–58, 2001.

[7] R. A. Botafogo and B. Schneiderman. Identifying aggregates in hypertext structures. In *Proc. 3rd Conference on Hypertext*, pages 63–74, 1991.

[8] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

[9] A. Z. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. *WWW6/Computer Networks*, 29(8-13):1157–1166, 1997.

[10] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *WWW9/Computer Networks*, 33(1–6):309–320, 2000.

[11] S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann, 2002.

[12] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. Tomlin, and J. Y. Zien. Semtag and seeker: Bootstrapping the semantic web via automated semantic annotation. In *Proc. 12th International World Wide Web Conference*, pages 178–186, 2003.

[13] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[14] D. Fetterly, M. Manasse, and M. Najork. Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages. In *7th International Workshop on the Web and Databases*, pages 1–6, 2004.

[15] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Proc. 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 150–160, 2000.

[16] D. Gibson. The site browser: Catalyzing improvements in hypertext organization. In *Proc. 15th ACM Conference on Hypertext and Hypermedia*, pages 68–76, 2004.

[17] D. Gibson. Surfing the web by site. In *Proc. 13th International World Wide Web Conference (Poster)*, page 496, 2004.

[18] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a webfountain: An architecture for very large-scale text analytics. *IBM Systems Journal*, 43(1):64–77, 2004.

[19] M. Henzinger, R. Motwani, and C. Silverstein. Challenges in web search engines. In *Proc. 18th International Joint Conference on Artificial Intelligence*, pages 1573–1579, 2003.

[20] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[21] R. Kumar, U. Mahadevan, and D. Sivakumar. A graph-theoretic approach to extract storylines from search results. In *Proc. 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 216–225, 2004.

[22] R. Kumar, J. Novak, P. Raghavan, and A. Tomkins. On the bursty evolution of blogspace. In *Proc. 12th International World Wide Web Conference*, pages 568–576, 2003.

[23] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the web. In *Proc. 27th International Conference on Very Large Data Bases*, pages 639–650, 1999.

[24] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the Web for emerging cyber-communities. *WWW8/Computer Networks*, 31(11–16):1481–1493, 1999.

[25] P. Pirolli, J. Pitkow, and R. Rao. Silk from a sow's ear: Extracting usable structures from the web. In *Proc. ACM Conference on Human Factors in Computing Systems*, pages 118–125, 1996.