Web Graph Similarity for Anomaly Detection

Panagiotis Papadimitriou
papadimitriou@stanford.edu

Ali Dasdan
dasdan@yahoo-inc.com

Hector Garcia-Molina
hector@cs.stanford.edu

(Last updated on 20th March 2009)

Abstract

Web graphs are approximate snapshots of the web, created by search engines. They are essential to monitor the evolution of the web and to compute global properties like PageRank values of web pages. Their continuous monitoring requires a notion of graph similarity to help measure the amount and significance of changes in the evolving web. As a result, these measurements provide means to validate how well search engines acquire content from the web. In this paper we propose five similarity schemes: three of them we adapted from existing graph similarity measures, and two we adapted from well-known document and vector similarity methods (namely, the shingling method and random projection based method). We empirically evaluate and compare all five schemes using a sequence of web graphs from Yahoo!, and study if the schemes can identify anomalies that may occur due to hardware or other problems.

1 Introduction

A search engine has two groups of components: online and offline. The online group processes user queries in real time and returns search results. The offline group collects content from the web and prepares it for the online part.

The offline group essentially consists of three main components: a crawler component to find and download web pages, a web graph component to create graphs out of pages and their links, and an indexer component to index pages using their content.

The quality of the search results produced by the online components depends on the "quality" of the data and structures generated by the offline components. And the quality of the offline components depends on a variety of issues: Did the crawler miss a "significant" number of important pages? Is the web graph an accurate representation of what was crawled? Is the index consistent with the crawl and the graph?

In practice, the quality of the offline component can be affected by a variety of what we call *anomalies*. For instance, a web host that is unavailable at crawl time may cause us to miss its content, as well as the content reachable from that host. Since the offline data is massive, spread over 1000s of machines and files, and gathered and manipulated over relatively long periods of time (hours to days), processor, network or software problems can corrupt parts of the data. Similarly, new policies put in place (e.g., to combat web spam or to for grouping virtual hosts into hosts) may have unexpected effects and lead to invalid data.

It is very hard to detect problems with the offline data simply by examining a single snapshot or instance. For example, how can one tell that an important part of the web is missing? Or that IP addresses were not properly grouped into hosts?

Because of the difficulty of identifying anomalies in a single data snapshot, it is more practical to identify anomalies based on "differences" with previous snapshots. The idea is to compute one or more similarity measures between two snapshots (of the crawl, the web graph or the index) and anomalies

will result in similarities that are too low (or too high). For instance, one similarity metric for the web graph may reflect how many hosts the two graphs share in common. If many hosts are missing from a new web graph, it probably indicates that something went wrong in the crawl or in the graph building. The similarity between graphs can also be used to tune the frequency of crawls. That is, if the similarity between one graph and the next is very high, it may indicate that it is unnecessary to crawl and build the graph so frequently. If the similarity is too low, we may need to crawl more often.

The challenge in this approach to anomaly detection is in developing similarity metrics that (a) can be computed in a reasonable time and in an automated way, and (b) are useful for detecting the types of anomalies experienced in practice. A metric that is too sensitive or sensitive to differences that do not impact the quality of search results will yield too many false positives. Similarly, a metric that is not sensitive enough will yield too many false negatives. There is, of course, also the challenge of selecting the proper thresholds that tell us when similarities are "too high" or "too low."

In this paper we focus on anomaly detection for the web graph component. In Section 3, we provide examples of anomalies that we target. They are similar to real anomalies we have observed at Yahoo!. Anomaly detection in other components is also as important and challenging but not covered here. However, note that crawl anomalies will often also be reflected as web graph anomalies, so our web graph work will also be helpful for dealing with many crawl issues. To the best of our knowledge, this is the first work in the area that addresses the anomaly detection problem for web graph component through the calculation of similarities or differences between consecutive snapshots.

We note that our work is empirical in nature. We take a variety of existing similarity schemes from other areas and apply them to our problem. The domain specificity or high time complexity of these schemes prevent us from applying them directly to web graphs in general and huge web graphs in particular. Hence, we modify and extend the schemes to make them work on the very large web graphs we have in our dataset.

To summarize, the main contributions of this paper are:

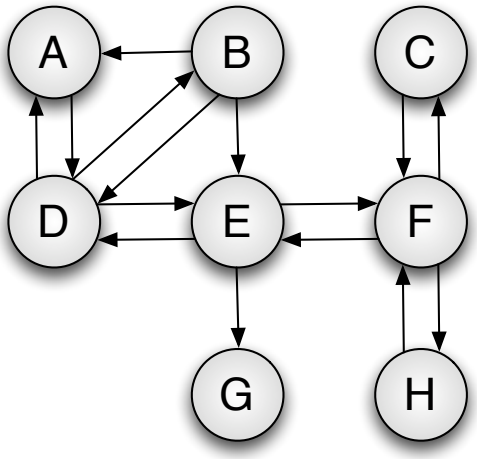
- We propose detecting anomalies through their impact on the web graph rather than their causes (Section 3).
- We propose using graph similarity for web graphs as a way to detect anomalies (Section 4);
- We present five scalable schemes for graph similarity (Section 5).
- We propose a locality sensitive hashing (LSH) function for web graphs in the context of our Signature Similarity scheme (Section 5.5);
- We provide experimental results using real web graphs from Yahoo!, and anomalies based on actual problems observed at Yahoo!, and compare all the presented schemes and discuss their pros and cons (Section 6).

2 Web Graphs

Web graphs are useful in many ways but their main purpose is to compute properties that need a global view of the web. PageRank is one such property, but there may be hundreds of other properties that need a global view. For example, the total quality of a host can be computed by aggregating the PageRank of its pages. These properties are used by the online search engine, but they also provide invaluable information on how the web is evolving.

Web graphs of pages (called *page graphs*) are also used to create many other types of graphs, some of which represent hierarchies (e.g., domain hierarchies) and some which represent subgraphs (e.g., language subgraphs). The resulting graphs are often more useful than the original page graphs due to their small size, which allows efficient processing. In our paper we focus on host-level web graphs (called *host graphs*), since they are extensively used in search industry. For some advantages of host-level graphs in link analysis of the Web, see [22].

A (host-level) *web graph* is a directed, weighted graph whose vertices correspond to active hosts of the web, and whose weighted edges aggregate the hyperlinks of web pages in these hosts. We represent



(a) Tiny Web Graph.

Machine	Vertex	Outlinks	PageRank
1	A	D	0.56
	B	A, D, E	0.43
2	C	F	0.51
	D	A, B, E	1.01
3	E	D, F, G	0.93
	F	C, E, H	1.29
4	G		0.41
	H	F	0.51

(b) Columnwise Representation.

Figure 1: The Tiny Web Graph and its representation. The graph on the left has 8 vertices, from A to H, and 15 edges. The representation on the right shows the vertices, their outlinks, their quality or PageRank scores, and how they are stored in a distributed system with 4 machines.

a web graph as $G = \{V, E\}$ where V is the set of vertices (hosts) and E is the set of weighted edges (aggregated hyperlinks). G has an edge (u, v) if there is at least one hyperlink from a web page of host u to a web page of host v. Each vertex and edge can have one or more weights. For the purpose of this paper, the weight $w(u, v)$ of an edge (u, v) is equal to the number of hyperlinks from pages of u to pages of v.

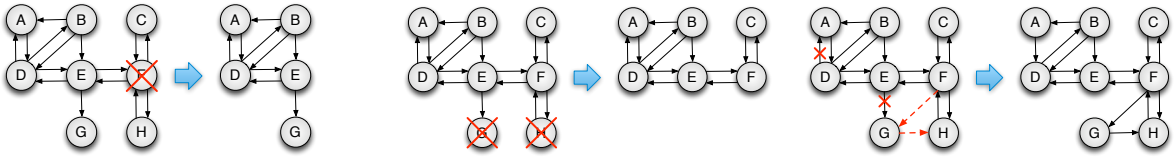
A *subgraph* G' of a web graph G is a directed, weighted graph whose vertices and edges are subsets of G. We represent subgraph G' as $G' = \{V', E'\}$, where $V' \subseteq V$ and $E' \subseteq E$. We call G' a *vertex-induced* subgraph of G if G' includes all the edges of E whose endpoints are in V'. In the rest of the paper, we refer to the vertex-induced subgraph of vertices V' simply as the subgraph of G with vertices V'.

Besides its topological features, a web graph has called *properties*. These properties can be numerical (scalars or distributions) or categorical (labels or lists). Some vertex properties that we will focus on are PageRank as a "quality" score (computed in a host graph [11]), the list of hosts pointing to it (called its *inlinks*), and the list of hosts it is pointing to (called its *outlinks*).

3 Potential Anomalies

Since our goal is anomaly detection, we now give examples of the types of anomalies we are interested in detecting. For illustration, we will use the Tiny Web Graph in Fig. 1(a) that contains 8 vertices and 15 edges. A search engine starts crawling from a web page of host A and discovers the rest of the hosts-vertices of the Tiny Web Graph. For each vertex v the search engine keeps three properties: name of the vertex, outlinks and quality $q(v)$, computed from its PageRank. The graph is stored in a column-oriented representation on a 4-machine computer cluster. The files on each machine correspond to vertex properties: the vertex name, outlinks, and quality. Each machine contains only a fragment (rows) of each file that corresponds to two vertices.

An anomaly occurs when the stored graph representation does not reflect the topology and the properties of the actual web graph at crawl time. An anomaly can be caused by either the search engine infrastructure, e.g., hardware failures, crawler bugs and web graph data management code bugs, or from problems in the public internet infrastructure, e.g., host failures and network outages. Although there are ways to handle and fix the problems that arise individually by each of these causes,



(a) Missing Connected Subgraph. (b) Missing Random Vertices. (c) Connectivity Change.

Figure 2: Examples of web graphs that are affected by anomalies.

the problem of monitoring all the infrastructure that is involved in the web graph creation and verifying its proper function is expensive and has no guaranteed overall solution. In this paper we focus on the effects anomalies have on web graphs rather than on the causes of anomalies. Our goal is either to detect an anomaly even if we are unaware of its cause, e.g., there is an unknown bug in the crawler, or quantify the effect of an anomaly if we are aware of it. For example, suppose that some graph data of the storage cluster are lost due to hard drive crashes. In this case, we want to know how much change is introduced due to this anomaly. If it takes lots of time and resources to build a web graph, it is useful to ensure that the graph is still "okay" before rushing to a costly rebuilding of the graph from scratch. Web graph similarity can help us make an informed decision in this case.

In the following paragraphs we provide a classification of anomalies into three categories based on their impact on the web graph representation with respect to the actual web graph. For each category we provide some anecdotal example causes.

Missing Connected Subgraph. Anomalies during crawling or machine failures in the storage cluster may result in a stored web graph that lacks a connected subgraph of the actual web graph. In Fig. 2(a) we show an example where during crawling the Tiny Web Graph host F is not reachable, e.g., due to the disruption of internet access. In this case the crawler will fail to discover the subgraph that is reachable through F and contains nodes C, F and H.

Missing Vertices. Suppose that machine 4 fails. There are obviously many fault tolerance mechanism that will recover the data on the failed machine but let us assume the worst case scenario that the data is lost. In this case we will lose all the information about vertices G and H as shown in Fig. 2(b).

Note that the consequences of a machine failure can be different based on how the vertices are distributed to the machines. If the distribution is such that each machine contains vertices sharing a common property, say, the same country, the failure will lead to the loss of part or all of this country's data. This case is similar to the removal of a subgraph from the web graph that we discussed above. However, if the distribution is random, the failure will lead to 'cuts' all over the web graph. The problem gets more serious if the missing vertices are important vertices in that they have very high rank in quality, or they are major authorities or hubs. For exampe, failure of machine 3 is more serious than failure of machine 4, since vertices E and F have higher PageRank values than G and H.

Connectivity Change. Connectivity change refers to anomalies where scalar graph properties such as the total numbers of vertices and edges do not change but for some vertices, their connectivity, i.e., the edges these vertices are adjacent to, changes. This anomaly may be caused by a variety of phenomena. For example, the crawler may disinterpret the hyperlinks of some web pages because of error encoding assumptions. Fig.2(c) shows an example where the crawler did not discover edges DA and EG and it invented two non-existent edges: FG and GH.

However, even if the hyperlinks are correctly collected by the crawler and the corresponding edges are correctly stored, there is no guarantee that the code that is used to access the web graph data is bug-free. For example, modern search engines use column-orientation in flat fragmented files to store graph data, since this approach has many advantages for storing and processing search engine data [3, 7]. However, this approach requires the development of custom code to access the graph data, e.g., to

fetch vertex names, to join vertices with edges, etc.. If there is bug in the code that joins vertices with edges the search engine will rely on a graph with altered edges.

A connectivity change is conceptually simple but is difficult to detect using scalar properties of web graphs. It may also have drastic consequences for a real web graph, since it alters the graph topology that affects the quality scores of the edges in case of PageRank.

4 Web Graph Similarity

4.1 Problem Formulation

We have a sequence of web graphs $G_1, \ldots, G_n$ built consecutively and we quantify the changes from one web graph to the next. We do this by computing one or more similarity scores between two consecutive web graphs, G_i and G_{i+1}. Similarity scores, when viewed along the time axis, create a time series. Anomalies can be detected by comparing the similarity score of two graphs against some threshold, or by looking for unusual patterns in the time series.

4.2 Similarity Requirements

A similarity function $sim(G, G') \in [0, 1]$ has value 1 if G and G' are identical, and value 0 if G and G' share no common features. The similarity function needs to satisfy the following requirements to be useful for our domain:

1. **Scalability**: The similarity function must be scalable to large web graphs or subgraphs. We call an algorithm *scalable* in our domain if its time complexity in practice is at most a *small* constant multiple of the web graph size. Due to ever growing size of web graphs, algorithms that are not scalable are simply impractical to run on web graphs.

2. **Sensitivity**: The similarity function must be more sensitive to changes in high-quality vertices and their edges. Changes in low-quality vertices are less important and should have a significant impact on the similarity computation only in cases where they affect a large proportion of the web graph vertices. For example, suppose that there is a specialized search engine for the stanford.edu domain that maintains the host graph $G(V, E)$ of the domain. We create one subgraph G' of G by removing the web host www.stanford.edu that is the highest-quality host of the original graph G. We also create another subgraph G'' by removing the lowest-quality host user.stanford.edu from the original graph G. If a similarity function satisfies the sensitivity requirement we expect that $sim(G, G') < 1$ and $sim(G, G'') \approx 1$.

3. **Coverage**: The similarity function must be sensitive to changes in both topology and the different properties of the web graph. Topological changes include introduction of new hosts, eliminations of existing hosts and changes in the hyperlinks of hosts web pages. As far as the changes in properties, we are mainly concerned about changes in the hosts' quality scores. For example, let $\dot{G}$ be a web graph that arises from G if we swap the hosts www.stanford.edu and user.stanford.edu. Note that the quality scores of the two hosts will also be swaped. Let also $\ddot{G}$ be the web graph that arises from graph G if we swap two medium-quality scores of G. Although all three web graphs are topologically similar, a similarity function with the coverage requirement should yield $sim(G, \dot{G}) < 1$ and $sim(G, \ddot{G}) \approx 1$.

The similarity computation schemes we propose are all scalable by our choice and design. We use experimental analysis over different anomaly scenarios to evaluate their sensitivity and coverage for real web graphs.

5 Computing Similarities

The problem of comparing graphs or computing their similarity has been an important problem with applications in many areas, from biological networks to web searching. For an overview, see the

book [6] and two web documents [21, 23], or search on the internet using the query "graph similarity", which returns many useful links in major search engines. Naturally, the diversity of the areas has created different approaches to graph similarity.

After carefully reviewing the families or types of proposed schemes, we identified some families as the most promising for our problem at hand. Within each family, we then developed a particular algorithm that was scalable and would be sensitive to anomalies with high coverage. We discuss these algorithms in Sections 5.1, 5.2 and 5.3.

In Sections 5.4 and 5.5, we propose two new approaches to graph similarity. They are inspired by the successful methods for document similarity, namely, the shingling method and the random projection based method. The latter relies on a novel application of a Locality Sensitive Hashing (LSH) function to web graphs.

In the following five subsections, we first describe the *family* of similarity schemes, then one or more *existing algorithms* in the family, and then our particular *proposed algorithm* adapted from that family. The families of graph similarity schemes that we considered inappropriate for our problem are briefly discussed in Section 7.

5.1 Vertex/Edge Overlap

This family uses the rule that "two graphs are similar if they share many vertices and/or edges". Although this approach seems to ignore the sensitivity and coverage requirements, it is effective in detecting some type of anomalies that we discuss in Section 6. Two main ways of computing this kind of similarity is graph edit distance and the Jaccard index [5] (as we apply to graphs). Graph edit distance [6] counts the number of some operations on vertices and edges to transform one graph to the other. The operations consist of insertions, deletions, and in the case of labeled graphs, renamings. The Jaccard index is one way to compute the overlap between two graphs. It is defined as the intersections of vertices/edges divided by the union of vertices/edges.

For this family, we defined the similarity of two graphs G and G' using a form of edit distance as

$$sim_{VEO}(G, G') = 2\frac{|V \cap V'| + |E \cap E'|}{|V| + |V'| + |E| + |E'|}. \quad (1)$$

We can compute this measure, say by scanning the V vertices (time linear on $|V|$) and checking if each occurs in V' (and similarly for E). If we assume that fetching a specific vertex or edge by name takes O(1) time, then this similarity computation takes $O(|V| + |V'| + |E| + |E'|)$ time.

The algorithm implementing this scheme is called the *Vertex/Edge Overlap* Algorithm (**VEO**).

5.2 Vertex Ranking

This family uses the rule that "two graphs are similar if the rankings of their vertices are similar". In this family, the vertices are usually ranked using their quality, and the similarity of rankings is usually computed using a rank correlation method such as Spearman's rho (denoted ρ). Although rank correlation is well known in the information retrieval field, its application to graph similarity appears to be new. A very related application, proposed in [4], is to the similarity of ranking algorithms.

The particular vertex ranking algorithm we use proceeds as follows. Let $G = (V, E)$ and $G' = (V', E')$ be the two graphs that we want to compare. For each graph we rank the vertices using their quality (scores), producing a sorted list of vertices. Then we find the correlation of the two lists using a modified version of Spearman's ρ.

The original formula for Spearman's ρ for comparing two ranked lists that are permutations of each other is

$$\rho = 1 - \frac{2\sum_i d_i^2}{n(n^2-1)/3} \in [-1, 1] \quad (2)$$

where n is the number of elements in each list and d_i is the difference between the ranks of the same element i in each list. Note that this comparison does not use any weights.

In our case, we have two constraints: (1) we want the rank correlation result to be sensitive to quality, and (2) we want to compute rank correlation for two lists that are not permutations of each

other. As such, inspired by [12], we revise the formula in Eq. 2 to a similarity measure as

$$sim_{VR}(G, G') = 1 - \frac{2 \sum_{v \in V \cup V'} w_v \times (\pi_v - \pi'_v)^2}{D} \quad (3)$$

where π_v and π'_v are the ranks of v in the sorted list for G and G', respectively, w_v is the quality of v, and D is a normalization factor that limits the maximum value of the fraction to 1. The resulting value of the similarity function is equal to 0 for perfect dissimilarity and 1 for perfect similarity (instead of -1 and 1 for the Spearman's ρ).

One subtleness is, what are the quality and rank of a vertex if it does not exist in one of the graphs? We resolve it as follows: If a vertex exists in both graphs, its final quality is the average of its qualities from both graphs; otherwise, its final quality is equal to its quality in whichever graph it exists. Moreover, if a vertex does not exist in one graph, say, G', its rank is equal to $|V'| + 1$. Vertices not in $|G|$ are handled analogously.

Given that vertex quality scores have been precomputed and that the vector of vertices for one graph fits in memory, we can compare two graphs with one scan of their vertex files and an expected running time $O(|V| + |V'|)$. If the second condition fails, the computation can still be done in linear time after the ranked lists are joined on disk based on vertex names.

The algorithm implementing this scheme is called the *Vertex Ranking* Algorithm (**VR**).

5.3 Vertex/Edge Vector Similarity

This family uses the rule that "two graphs are similar if their node/edge weight vectors are close". For this family, the weights usually represent quality. More formally, if $v_1, v_2, \ldots$ are the vertices shared by two graphs G and G', then we build vectors Q and Q' for these graphs, respectively, where $Q[i]$ is the quality $q(v_i)$ of vertex v_i. Then we compare the two vectors by computing the average difference between all $Q[i]$, $Q'[i]$. For some works that use this approach, see [6, 20, 24].

A slight variation on this approach is proposed by [19], which computes the similarity of vertex degree distributions. We believe this variation can readily be extended to other graph properties such as those discussed in [10]. Similarly, if one has edge properties, it is possible to construct edge vectors and compare them.

For our particular similarity measure, we compare edges, giving each edge a weight that captures the local topology. As a base, for edge (u, v), we compute weight $\#outlinks(u, v)$ as the number of outlinks from u to v, where these vertices can be pages, hosts, or domains. Then we compute $\gamma(u, v)$ to capture the relative importance of this edge to other edges leaving node u:

$$\gamma(u, v) = \frac{Q_u \times \#outlinks(u, v)}{\sum_{\{v':(u,v') \in E\}} \#outlinks(u, v')} \quad (4)$$

where Q_u gives the quality of u.

This approach assigns higher weights to edges that start from high quality vertices. Using these γ weights, we calculate the similarity between two graphs G and G' as

$$sim_{VS}(G, G') = 1 - \frac{\sum_{(u,v) \in E \cup E'} \frac{|\gamma(u,v) - \gamma'(u,v)|}{max(\gamma(u,v), \gamma'(u,v)}}{|E \cup E'|} \quad (5)$$

where the denominator is a normalization factor.

In cases where an edge (u, v) appears in G but not in G', we let $\gamma'(u, v) = 0$ to set the numerator to one, resulting in a lower similarity between G and G'. The case where (u, v) is in G' but not in G is handled analogously.

This similarity computation runs in $O(|E| + |E'|)$ time.

The algorithm implementing this scheme is called the *Vector Similarity* Algorithm (**VS**).

5.4 Sequence Similarity

This family uses the rule that "two graphs are similar if they share many sequences of vertices and edges, i.e., short paths". The algorithms in this family are used to compare objects that are naturally sequenced, e.g., documents that consist of a sequence of words. For example, shingling [5] is frequently used to detect near-duplicate web pages [15].

Because sequence comparison algorithms are efficient and can operate on large inputs, we want to consider them as candidates for our problem. Thus, here we use a sequence comparison scheme, shingling in particular, to compare two graphs. The main challenge for us is converting the graphs into linear sequences that can then be compared using shingling. As far as we know, our proposal is the first application of shingling to graph similarity. However, a related work is [14], where shingling was applied to the detection of large dense subgraphs.

We start the description of our Sequence Similarity Algorithm by reviewing the base shingling scheme of [5]. We then provide a function ϕ that transforms a graph into the input required by shingling, i.e., into a sequence of tokens. The goal for our function ϕ is to produce a sequence that is strongly correlated with the features of the graph so that the sequence similarity reflects graph similarity.

The intuitive definition of document similarity in the shingling method is that two documents d and d' are similar if they have in common many subsequences of words. Actually, the shingling method estimates the ratio of the common subsequences of a fixed length k that appear in two documents over the size of the union of such subsequences in both documents. In our application of the algorithm to graph similarity, the value $k = 3$ was selected after a small series of experiments.

The shingling method works as follows. First, we convert every document into a canonical sequence of tokens $T =< t_1, \ldots, t_n >$ using standard information retrieval techniques like tokenization, case folding, etc. From this canonical sequence, we extract all the subsequences of k tokens and fingerprint them using a fingerprinting mechanism that maps each token to a number in a set U. Each of the $n-k+1$ fingerprints so obtained are called *shingles*. We use $S(T)$ to represent the set of shingles resulting from the sequence of tokens $T =< t_1, \ldots, t_n >$ for a document d. The similarity of two sequences T and T', and consequently the similarity of the respective documents d and d', is computed by the ratio $\frac{S(T) \cap S(T')}{S(T) \cup S(T')}$, which is the Jaccard index.

For storage and performance reasons, the shingling scheme actually uses an unbiased estimate to approximate the ratio above. We choose m random permutation functions $\pi_i : U \rightarrow U, 1 \leq i \leq m$ and apply them to the elements of $S(T)$. From the i-th permutation $\pi_i(S(T))$ of $S(T)$ we keep the minimum element $min(\pi_i S(T))$, which is called the i-th *minvalue*. In this way, we end up with an m-dimensional vector $\mathbf{w}$ for each document. The similarity estimate for documents d and d' is equal to the percentage of agreeing entries in vectors $\mathbf{w}$ and $\mathbf{w}'$.

We now turn to the description of our proposed function ϕ that has a graph as its input and produces a sequence elements out of it. Hence, ϕ is defined as $\phi(G) \rightarrow < t_1, t_2, \ldots >$. The transformation of a graph into a sequence of elements implies essentially some loss of information about the graph. For instance, we can convert a graph to a sequence of sorted vertices on the name of each vertex. In this case, we miss all the information regarding the connectivity among vertices. Since we are concerned with identifying significant changes in the topology of the graph (the coverage requirement of Section 4), we have to produce a sequence that will be explicitly affected by topological features of the graph. In general, the problem of converting a graph into a serial representation such that strongly correlated vertices are placed next to each other is known as the *graph seriation problem* [20].

The problem of searching for a serial ordering of the vertices to maximally preserve the edge ordering has exponential complexity [20]. However, there are many approximate solutions to this graph seriation problem. Using the example of Fig. 1, we can convert the graph into a sequence of vertices ordered by quality, i.e., F, D, E, A, F, H, B, G. However, this sequence satisfies the sensitivity requirement of Section 4 but violates the coverage requirement. To satisfy both, we use a solution as detailed next.

Our Sequence Similarity algorithm to compute ϕ uses the *walk algorithm*. The walk algorithm visits the vertices in their quality order but following only the edges of the graph, i.e., there is no jumping from one vertex to another unless all neighbors of a vertex are visited. This algorithm is similar to the one proposed in [20]. More formally, the walk algorithm works as follows: Start off by visiting the

vertex associated with the highest quality, and repeat the following process: 1) Among the unvisited neighbors reachable from the currently visited vertex, visit the neighbor with the highest quality; 2) If the currently visited vertex does not have any unvisited neighbors, jump to the vertex that has the highest quality among all the unvisited vertices.

To illustrate this algorithm, with Fig. 1, we start with the highest quality vertex F. Then we do a traversal following the links that lead to the highest quality (unvisited) vertices. The highest quality neighbor of F is E, so we visit it next. From there we visit E's highest quality neighbor, D. After D we visit A. Since A's neighbors have all been visited, we jump to the highest quality unvisited vertex, C or H. We pick C at random. From C we cannot reach unvisited vertices, so we again jump, this time to H. From H we jump to B and then we jump to the last unvisited node G. The final serialization is F, E, D, A, C, H, B, G.

Given the quality values of the vertices, the walk algorithm runs linearly in the size of the graph. After the walk algorithm, the calculation of shingles and of the vector with the *minvalues* takes time proportional to the number of vertices of the graph.

The algorithm implementing this scheme is called the *Sequence Similarity* Algorithm (**SeqS**).

5.5 Signature Similarity

This family is based on the rule that "two objects are similar if their signatures are similar". Instead of converting a graph into a sequence, we convert it into a *set* of features, which are then randomly projected into a smaller-dimensionality feature space (signatures) for comparison.

There are many ways to compare sets of features, but here we focus on a scheme called *SimHash* originally developed for high dimensional vector comparison [8] and is applied to documents comparison [15]. Again, our challenge is in converting our graphs into appropriate sets of features to be input into the SimHash algorithm.

We start by reviewing the *SimHash* algorithm [8]. The algorithm provides a technique to map high-dimensional vectors to small-sized fingerprints. *SimHash* has the property that similar vectors are mapped to similar fingerprints with respect to the Hamming distance between them. As such, it is one of the locality sensitive hashing (LSH) methods. Using *SimHash*, the problem of document similarity is reduced to the similarity of their fingerprints.

The $SimHash$ algorithm works as follows. Initially, a document d is transformed to a set of weighted features $L = \{(t_i, w_i)\}$ where feature t_i is a token of d and w_i is its frequency in d. Tokens are also obtained as in shingling and appear only once in set L. This weighted set can be viewed as a multidimensional vector. The dimensions of the vector space are defined by the distinct features that can be found across all documents. Using this weighted set as input, *SimHash* provides a b-bit fingerprint as follows. Every token t_i is randomly projected into a b-dimensional space by randomly choosing b entries from $\{-w_i, +w_i\}$. We perform this projection using a hash function to obtain the digest for every token t_i. The length of the digest must be equal to b bits (or greater, since we can crop it and reduce it to b bits). We associate $+w_i$ with every 1 and $-w_i$ with every 0 of the digest. The final fingerprint of the document is produced by summing up all the b-dimensional vectors to a single vector h of length b and then converting it to a bit string as follows: the i-th entry of h is set to 1 if the entry is positive and 0 if it is negative. The similarity of two documents with fingerprints h and h' is assumed to be equal to the percentage of agreeing bits in h and h'.

The size of the final fingerprint depends on the size of the digest of the hash function we apply to every token. Given the amount of information that we reduce to a small fingerprint we picked the size of this fingerprint as large as practical. The largest available digest in the library of the programming language we used (see Section 6) was 512 bits.

Intuitively, two high dimensional vectors are similar for this method if their respective coefficients have the same direction and the same relative length compared to the other coefficients. Features with small weights play a less important role than features with large weights.

We can view SimHash as a method that provides a similarity estimate between two weighted sets L and L' as

$$sim_{SimHash}(L, L') = 1 - \frac{Hamming(h, h')}{b} \tag{6}$$

where h and h' are the b-bit vectors corresponding to L and L', respectively, and $Hamming(h, h')$ is equal to the number of bit positions in h and h' for which the corresponding bits are different. Note that the resulting value is equal to 0 for perfect dissimilarity and 1 for perfect similarity.

We now move to the problem of defining an appropriate function ϕ that transforms a graph into a weighted set, i.e., $\phi : G \rightarrow L$. Function ϕ returns for each graph a set of weighted features that represent the properties of the graph. Using function ϕ, the similarity between two graphs is then expressed as

$$sim_{SS}(G, G') = sim_{SimHash}(\phi(G), \phi(G')) \tag{7}$$

which is in turn given by Eq. 6.

In our case, we want the features to strongly reflect the topological features of the web graph (the coverage requirement). The features we select are the vertices and the edges of the graph. We assign to a vertex v its quality, and to an edge (u, v) the quality of vertex u normalized by the number of its outlinks. Note that this normalization is similar to the one in Section 5.3. For example, the set of weighted features for subgraph G′ containing only vertices C, F and H of Fig. 1 is:

$$\begin{aligned} L(G') \quad = \quad & \{(C, 0.51), (CF, 0.51), (F, 1.29), (FC, 1.29 \times 0.5), \\ & (FH, 1.29 \times 0.5), (H, 0.51), (HF, 0.51)\}. \end{aligned} \tag{8}$$

Since the features of the weighted set are vertices and edges of the graph, we penalize/reward the absence/presence of such common features among different graphs. The use of quality as a weight attributes to every vertex and edge the proper significance from a search engine perspective. The disappearance of a high-quality vertex of a web graph in a new snapshot is penalized more than the disappearance of low-quality vertices.

Assuming again that the quality scores of the vertices of the graph are precomputed, the calculation of its fingerprint requires just one scan of its edge file. This takes time $O(|E|)$. Then, given the fingerprints of two graphs we can compare them in $O(b)$ time, which is $O(1)$ as b is a constant.

The algorithm implementing this scheme is called the *Signature Similarity* Algorithm (**SS**).

6 Experiments and Results

We performed experiments to evaluate the similarity measures and algorithms presented in Section 5. The experimental section is divided into three subsections.

In Section 6.1, we describe the dataset that we used for our experiments and introduce some notation. We also provide some basic statistics about our dataset.

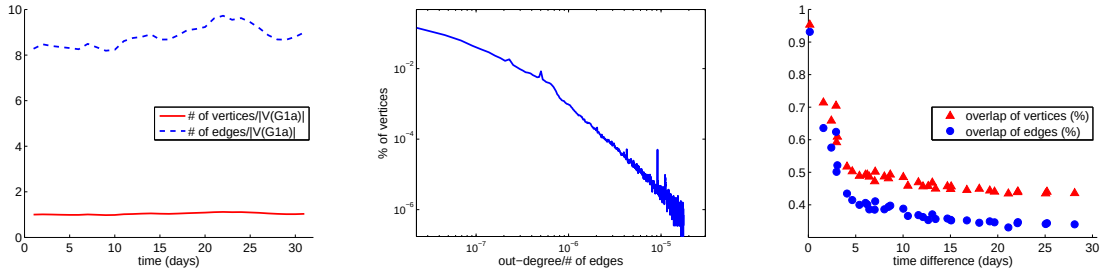
In Section 6.2, we experimentally show how similarity varies over time for the different similarity measures and algorithms. This analysis helps us determine the thresholds that identify "significant" changes in a graph.

Finally, in Section 6.3, we evaluate how successful the algorithms are in detecting anomalies. In particular, we check whether significant changes in a web graph affect its similarity score relative to its predecessor graph so that an anomaly can be detected.

6.1 Dataset and Setup

For our experiments, we selected a sequence of host web graphs generated by the Yahoo! search engine over a month long period in 2007. These graphs come from crawls that were performed a few times a day to track some "important" web sites with fast changing content such as news sites. Anomaly detection in such graphs is more challenging than in regular web graphs: the fast evolution of the graph, e.g., with addition of new vertices and edges, with elimination of old edges, etc., makes it difficult to determine whether changes in the crawled web graph reflect changes in the actual web graph or are caused by an anomaly. We also studied anomaly detection for other kinds of host web graphs (for some other sets of "important" web sites) as well as for web graphs at different granularity levels such as web graphs of pages, domains, or countries. The results are analogous to what we present in this paper and are not discussed here.

We refer to a web graph obtained on the X-th day of the month as GX. Since for each day we had multiple graphs, we append lower-case letters to distinguish the ones we chose to use. For example,



(a) Average number of vertices/edges. (b) Outlinks distribution for G29b. (c) Percentage overlap in vertices/edges.

Figure 3: Statistics of the host web graphs in our dataset. The overlap computation was done using the Jaccard index. The time difference between two graphs refers to the difference between their build times.

the web graphs built on the 18th day are $G18a, G18b, G18c, \ldots$ where $G18a$ was the first graph of the day, $G18b$ the second, and so on. We chose only the first two for our experiments.

In Fig. 3(a), we show the size of our web graphs on a daily basis. The average size was around several tens of millions. For each day X, we report the number of average number of vertices and edges in graphs $GXa, GXb, \ldots$ normalized by the number of edges of the first graph of our dataset $V(G1a)$.

In Fig. 3(b), we plot the out-degree distribution of vertices of the web graph $G29b$. Other web graphs had similar distributions. Both axes are in logarithmic scale. The out-degree values are normalized by the total number of $G29b$ edges. The plot confirms that the number of outlinks per host follows a distribution similar to the power law distribution. We observe that almost 20% of hosts have exactly one outlink whereas a few hosts have a huge number of outlinks. We cannot see how many hosts have no outlinks because of the logarithmic scale of the plot. The percentage of such hosts was approximately 40%.

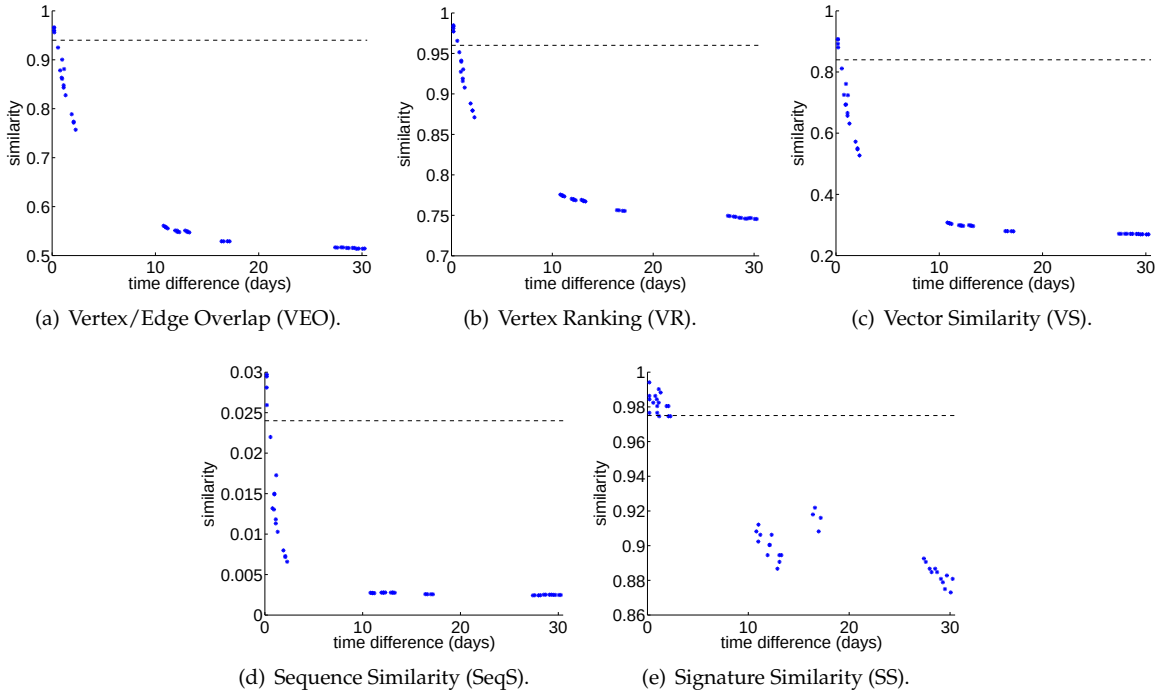
In the last figure, Fig. 3(c), we plot the evolution of the web graphs over time. This figure shows the percentage of overlapping vertices and edges between graphs obtained x days apart, as a function of x. Note that x has decimal values, e.g., the time difference between graphs GXa and GXb can be 0.3 days. We calculated the percentages of overlapping vertices for web graphs G and G' with the Jaccard index $\frac{|V \cap V'|}{|V \cup V'|}$. We used a similar formula for the edge overlap computation. [1]

In this figures, plots for both vertices and edges are similar. The decrease in overlap as time increases shows how graphs naturally diverge over time. Web graphs obtained on the same day have approximately 90–95% of their vertices and edges in common. The overlapping percentage of vertices reduces to 50% for graphs obtained 5 days apart and then it reduces slowly to approximately 45% for graphs obtained 30 days apart. The edges overlap curve drops to 40% in 5 days and to 35% in 30 days. The fast change of the web graph in the 5 day period is due to the fast content changes of the web sites that are tracked, e.g., the edge between www.cnn.com and www.yahoo.com exists as long as the article about the change of the Yahoo! CEO appears in some www.cnn.com web site and disappears when CNN removes this article. The change is much smaller after 5 five days, since the vertices and the edges of the 50% and 40% overlap correspond to more permanent hosts and hyperlinks among them, e.g., there is a permanent edge from www.cnn.com to weather.cnn.com.

The Yahoo! search engine has reported no anomalies in the software and hardware used to obtain these web graphs. Also, since all of them were successfully used by the search engine, it is reasonable to assume they had no anomalies. Thus, we may consider the information in the figures 3(a) and 3(c) as a reliable evidence of the evolution of the web graphs over time.

We performed the calculation of all the similarity measures locally using a single machine with CloverTownLV 1.86 GHz processor and 6GB RAM memory. We implemented all the algorithms in Perl

[1] The ratio of shared features between different graphs can also serve as a similarity measure between them. We explored this similarity measure in Section 5.1.



(a) Vertex/Edge Overlap (VEO). (b) Vertex Ranking (VR). (c) Vector Similarity (VS).

(d) Sequence Similarity (SeqS). (e) Signature Similarity (SS).

Figure 4: Similarity of web graphs as a function of time difference between their build times.

and Java. Each similarity computation for one pair of graphs took approximately 20 minutes (wall time), independent of the algorithm used. The running times were dominated by the input/output time spent during the linear scan over the edges file, which is why the running times were similar for all schemes. For this reason we do not discuss performance further.

6.2 Definition of Similarity Threshold

A similarity threshold t for an algorithm A is used to identify an anomaly. If two graphs that are expected to be similar, e.g., generated a day apart, have similarity less than t, then we will suspect an anomaly has corrupted one of the graphs. To determine similarity thresholds automatically and in a statistically sound way, we use in our production implementation both non-parametric and parametric methods from time series forecasting, e.g., see [13]. However, the discussion of these methods is beyond the scope of this paper. Here, we use a simple method with a fixed threshold t for each algorithm that works very well in practice.

To understand how to set the threshold t, we studied the similarity between graphs as they evolved over time. We measured the similarity of such web graphs with the goal of checking whether the presented similarity schemes could quantify the changes of the web graphs over time. These changes are imposed by the natural evolution of the web and grow as time passes. This fact is studied in [9, 18] and is also confirmed by Fig. 3(c). In particular, if we have a sequence of web graphs generated from crawls that were obtained sequentially over time, we expect the similarity to decrease over time. That is, in a web graph sequence we expect the following similarity to hold:

$$sim(G_i, G_j) > sim(G_i, G_k) \text{ if } i < j < k. \tag{9}$$

Thus, while the similarity between graphs generated close in time provides an estimate about the acceptable similarity score between consecutive web graphs, the similarity between graphs generated further apart in time provides evidence about the similarity score between web graphs that are not similar. Therefore, it is possible to fix threshold t for an algorithm A, provided that A yields signifi-

cantly higher similarity scores for graphs obtained close in time compared to graphs obtained further apart.

We experimentally tested which similarity measures have this discriminative feature. We applied all the similarity algorithms of Section 5 to measure the similarity among all pairs of 10 web graphs in our dataset. The web graphs included 5 pairs of consecutive graphs to get a good estimate of the similarity of two web graphs obtained close in time.

We present the results for all five algorithms over all pairs of graphs in our set in Fig. 4. Each plot focuses on one algorithm, e.g., Fig. 4(b) is for Vertex Ranking. A point in a plot shows the similarity between graphs obtained x days apart. Note that x has decimal values as in Fig. 3(c). All algorithms satisfy the inequality of Eq. 9 for every graph triplet. The points in the Signature Similarity plot that show lower similarity score for graphs closer in time correspond to disjoint pair of graphs, e.g., the similarity $sim_{SS}(G1a, G18a)$ between graphs that are 17 days apart is higher than the similarity $sim_{SS}(G18a, G30a)$ between graphs that are 12 days apart. However, $sim_{SS}(G1a, G17a) > sim_{SS}(G1a, G18a)$ and $sim_{SS}(G1a, G18a) > sim_{SS}(G1a, G30a)$.

The plots also show horizontal dotted lines at different values. These values represent our choices of threshold t for the different methods. Our rule for setting thresholds t was as follows. We select the smallest similarity between any pair of consecutive web graphs without anomalies, e.g., GXa, GXb or GYb, GYc, and we set t to roughly that value. Such a threshold will be below the similarity value of two graphs with small differences such as two consecutive graphs. However, web graphs that are significantly less similar than two consecutive ones, e.g., web graphs obtained more than a couple days apart, should have a similarity value below the threshold t. Although the last observation is a desirable feature for all the similarity algorithms and not a requirement that we took into consideration in the definition of threshold t, we see that all algorithms have it. In particular, in all plots the similarity scores between graphs that are obtained further apart lay far below the corresponding threshold lines.

Although we are concerned with relative similarity values for the definition of threshold t, observe that Sequence Similarity yields very small similarity values (Fig. 4(d)) and that Signature Similarity (Fig. 4(e)) yields values closer to 1, which is the similarity of two identical graphs. This means that Sequence Similarity is more sensitive to the changes that occur over time, while Signature Similarity is more tolerant of these natural changes.

6.3 Detecting Anomalies

In this section we try to address the following question: *Can our similarity measures be used to detect common anomalies and quantify their extent (as discussed in Section 3)?*

In the following experiments, we select pairs of consecutive web graphs GXa and GXb of the same day X and we inject different anomalies into GXb to get GXb'. We study the impact of the injected anomalies on the similarity between the two web graphs and the sensitivity of the different algorithms to anomalies. Our goal is to see whether we can detect the anomalies in the corrupted GXb' and estimate their extent, using the similarity score between GXa and GXb' and the threshold t from Section 6.2. Note that we are comparing GXb' to GXa, not to GXb. Graph GXb is the one that has been corrupted and in practice is not available in its uncorrupted form.

The anomalies that we study are missing connected subgraph (Section 6.3.1), missing random vertices (Section 6.3.2) and connectivity change (Section 6.3.3). Our results for each kind of anomaly have two parts:

- We show how each similarity measure captures the effects of an anomaly at different extents. This part of our results provides an insight to the sensitivity and the coverage of each similarity measure that is independent from the application where the meaures will be used.

- We also discuss whether a corrupted web graph is acceptable or not and decide whether a similarity measure is good or bad at detecting each anomaly. This part of our results is application dependent and our conclusions are not necessarily valid in different applications. In our case, we want to the use a newly obtained web graph to calculate vertex properties such as PageRank that are used in the online components of a search engine, e.g., in results ranking. Since the "important" hosts web graph evolves fast over time, we prefer to use a newly obtained and slightly

corrupted web graph rather than relying on an uncorrupted old one. For example, we prefer using slightly corrupted PageRank values that result in correct ranking for most of the fresh news articles rather than relying on perfect PageRank values that capture only older articles.

6.3.1 Missing Connected Subgraph

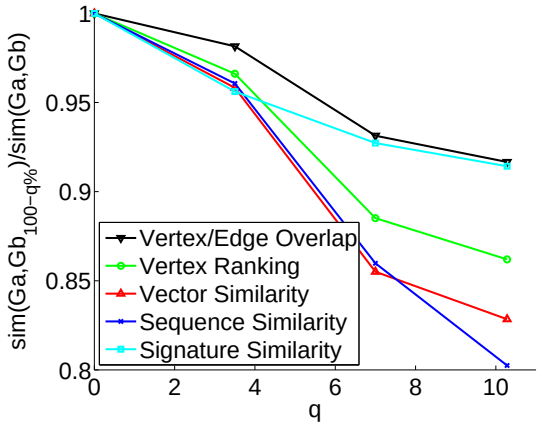


Figure 5: Similarity of a web graph with its corrupted versions due to missing connected subgraphs. The x-axis (log-scale) shows the percentage of the missing vertices and the y-axis shows the ratio of the similarity scores.

We use five pairs of consecutive graphs to simulate the anomaly of missing a connected subgraph: $G1a - G1b$, $G18a - G18b$, $G29a - G29b$, $G30a - G30b$ and $G31a - G31b$.

Let a consecutive pair be GXa-GXb. We created three corrupted versions of every GXb by eliminating connected subgraphs that account approximately for 3.5%, 7.0%, 10.5% of its vertices. We refer to the resulting web graphs as $GXb_{100-q\%}$, where q is the percentage of the eliminated vertices of GXb. To eliminate highly connected subgraphs, we removed vertices that belong to specific country domain(s). To create the corrupted version with 3.5%, 7%, 10.5% of missing vertices, we removed vertices of two, one, and three country domains, respectively, We selected domains from the top ranked domains when we sorted them in decreasing order of their contribution to the total number of vertices.

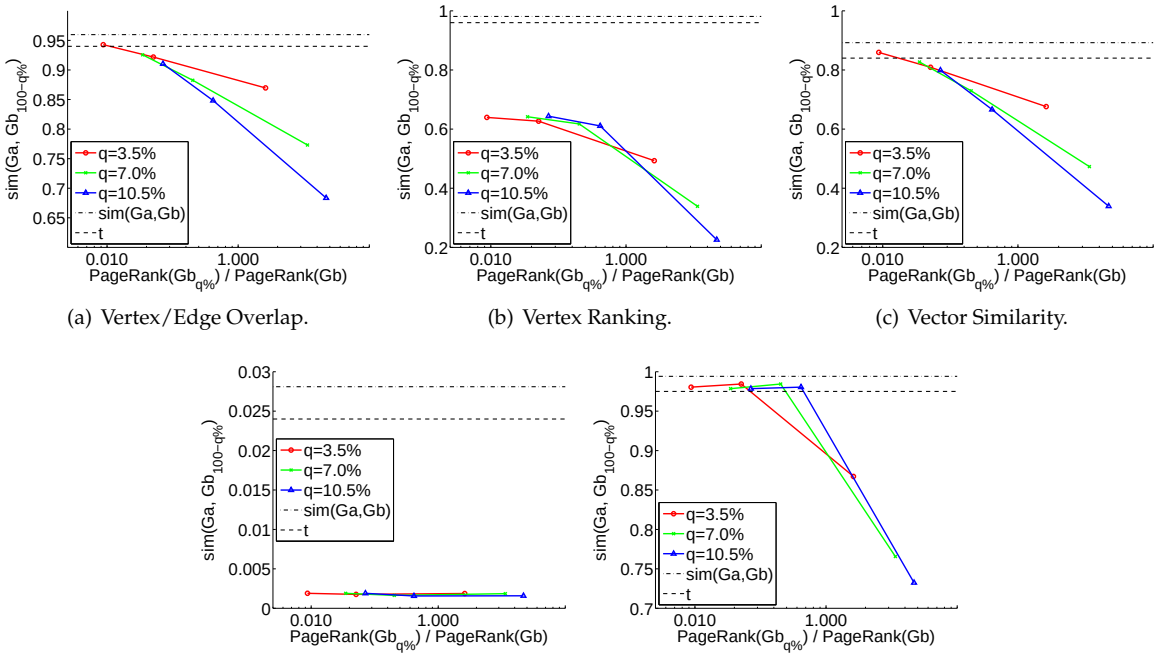
Note that given that consecutive web graphs have an overlap in 90–95% range (see Fig. 3(c)), we may well consider the removal of the small subgraph (3.5%) as a non-significant anomaly that has a small effect on the rest of the web graph. We consider the eliminations of greater subgraphs as more significant anomalies.

We calculated the similarity of every corrupted version of GXb with the corresponding GXa and compared the result with the similarity of the original web graphs of the pairs, as shown in Fig. 5. Each point on a plot shows the average similarity of an algorithm over the five different pairs. The deviation of similarities around the average is small for all algorithms and we do not report it here to make the plot easier to read.

We observe that the plots of all algorithms decrease monotonically with the extent of changes, but their slopes vary. The Vertex Ranking plot has the steepest descent while the Vertex/Edge Overlap plot decreases more gradually than that of any other algorithm does.

For all the algorithms, the similarity of GXa with $GXb_{96.5\%}$ is very close to the corresponding threshold, either below or above it. The comparisons with the other corrupted versions yield similarity scores that are clearly below the thresholds except from Sequence Similarity. The values of $sim_{SeqS}(GXa, GXb_{93\%})$ are very close to the threshold of Sequence Similarity and make it doubtful whether this scheme would always detect the elimination of a significant connected component of the web graph.

In summary, all the algorithms except Sequence Similarity successfully detect this type of anomaly.



(a) Vertex/Edge Overlap. (b) Vertex Ranking. (c) Vector Similarity.

(d) Sequence Similarity. (e) Signature Similarity.

Figure 6: Similarity of a web graph with its corrupted versions due to missing random vertices. The x-axis shows the percentage of the missing vertices and the y-axis shows the ratio of the similarity scores.

6.3.2 Missing Random Vertices

We used five pairs of consecutive graphs $G1a - G1b$, $G18a - G18b$, $G29a - G29b$, $G30a - G30b$, $G31a - G31b$, and we simulated the removal of random vertices from the GXb graph of each pair. We experimented with different numbers of vertices and vertices of different significance. First, we discuss how we selected the vertices we removed and then we provide the results of our experiments.

We removed from the graph GXb exactly 3.5%, 7.0% and 10.5% of its vertices to create the graphs $GXb_{96.5\%}$, $GXb_{93\%}$ and $GXb_{89.5\%}$, respectively. However, for each percentage, we removed vertices of low, medium, or high quality, where each case is denoted by appending the letters L, M, or H to the name of the resulting graph. For example, we removed three different sets of 7.0% vertices of the web graph $G29b$ to create $G29b_{93\%L}$, $G29b_{93\%M}$ and $G29b_{93\%H}$.

We used *reservoir sampling* to obtain uniformly random samples of fixed size. To bias to the selection of vertices to form $GXb_{q\%}$ with different aggregate quality scores, we performed the following process: we sorted the vertices of GXb in the descending order of their quality scores and divided the resulting list into three subsets. The high, medium, and low quality subsets contained n, $2n$, and $4n$ vertices of GXb so that $n + 2n + 4n = |V_{GXb}|$. The increasing sizes of subsets were motivated by the power law distribution of the quality scores [1]. Finally, we removed vertices with the given percentages from each subset separately.

Before presenting our experimental results, we comment briefly on which corrupted graphs were considered to have significant anomalies. On the one hand, we consider the elimination of high quality vertices as a significant anomaly for all different percentages of the removed vertices (3.5%-10.5%). Note that only 3.5% vertices of high quality account for approximately 17% of the aggregate quality score of the web graph. On the other hand, the elimination of medium and low quality vertices should affect the similarity score, but we believe the resulting graph should be considered as not corrupted. Due to the power law distribution of the quality scores, both low and medium quality vertices of any percentage (3.5%-10.5%) account for a small portion of the aggregate quality score of the whole graph. In the worst case of removal of 10.5% vertices of medium quality, the removed quality scores account

for approximately 6% of the aggregate quality score of the web graph. Given that the overlap of vertices and edges between consecutive graphs with no anomalies is in 90–95% range, e.g., see Fig. 3(c), we can still use the resulting web graph without major problems.

Fig. 6 shows our experimental results. We used the same corrupted versions of the web graph GXb to test the different algorithms. Each plot again looks at one similarity algorithm. The x-axis of each plot shows the ratio of the aggregate quality score of the corrupted subgraph $GXb_{q\%Y}, Y \in \{L, M, H\}$ to the aggregate quality score of GXb. The y-axis shows the similarity of a corrupted version of GXb with GXa. There are three plots in each graph, each of the them corresponds to the three different percentages q of removed vertices (3.5%, 7.0%, and 10.5%). There are also two additional straight dashed lines that show the similarity score of GXa with the original GXb and the threshold t of each algorithm for comparison purposes.

We see that Vector Similarity, Vertex/Edge Overlap, and Signature Similarity depend on the significance of changes in the desired way. They give acceptable or nearly acceptable similarity values for changes that affect only insignificant vertices (with Low and Medium quality scores). We see in Fig. 6(c) and in Fig. 6(a) that Vertex Similarity and Vertex/Edge Overlap give similarity scores above or slightly below the threshold for the comparisons of GXa with all corrupted versions of GXb with the missing vertices of Low and Medium quality except from $GXb_{10.5\%M}$. Signature Similarity, shown in Fig. 6(e), gives similarity scores above the threshold for all such comparisons. Although we expected this behavior from Vertex Similarity and Signature Similarity, as they consider explicitly weights for edges and vertices associated with the quality scores of vertices, Vertex/Edge Overlap seems to perform unexpectedly well. We had expected that a similarity measure like Vertex/Edge Overlap that considers only the overlap of features between two graphs would not discriminate between the removal of significant and non-significant vertices. However, the performance of Vertex/Edge Overlap is determined by the way we selected the removed vertices. When we selected vertices with high quality score, we implicitly selected vertices with large number of edges. Since the number of edges is 10:1 to the number of vertices the calculation of the ratio of union to intersection of vertices and edges is dominated by the corresponding ratio of edges. If we had performed the random selection of the subgraph using random selection of edges, Vertex/Edge Overlap would not have been as successful.

On the contrary, Vertex Ranking and Sequence Similarity showed undesired high sensitivity even in small changes. We see in Figures 6(b) and 6(d) that these algorithms give similarity scores below the threshold for all corrupted versions of GXb. These versions include web graphs with only minor changes such as missing 3.5% of low quality vertices of the original graph. This sensitivity would not allow us in a real case to use a web graph where 3.5% of insignificant vertices were missing, even though such a web graph includes most of the desired features to be used by a search engine.

To sum up, the experimental results of this subsection show that Signature Similarity has the desired sensitivity at anomalies of different significance, and that the behavior of Vector Similarity and Vertex/Edge Overlap is close to the desired one. In contrast, Vertex Ranking and Sequence Similarity fail to discern significant from insignificant anomalies.

6.3.3 Connectivity Change

We simulated a connectivity change by exchanging outlinks between vertices. In particular we assumed that links are stored by column as discussed in Section 3: there is one column for the first outlink of each vertex, another column for the second outlink and so on. To create a corrupted web graph we skipped the first row of the i-th outlinks column, changing the i-th outlink of all vertices in the graph. For example, if we skip the 3rd outlinks column in the Tiny Web Graph of Fig.1 we will replace edges DE, EG and FH with EE (self-edge that we can ignore), FG and GH respectively.

We used five different pairs of consecutive graphs, each pair obtained on a separate day. We simulated the same anomalies in each pair. Let a pair of consecutive subgraphs be $GXa - GXb$. We created four different corrupted versions of GXb by skipping the rows in the i-th outlinks column. Let us call such a column a "corrupted column". We named the resulting corrupted graphs GXb_{RSi}, $i = 1..4$, where i indicates the index of the corrupted column.

Skipping a single row seems like a simple anomaly but it can result in drastic changes in the topology of a web graph. For example, corrupting the first column affects approximately 60% of the hosts,

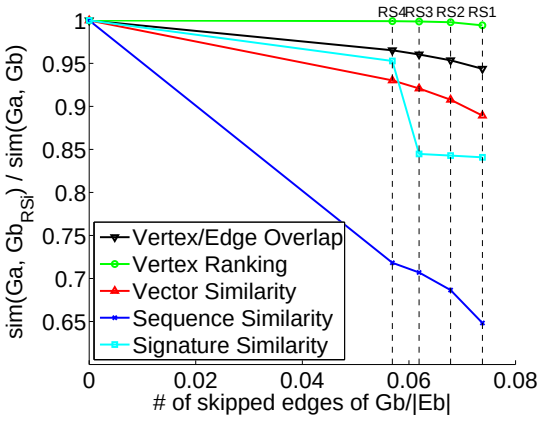


Figure 7: Similarity of a web graph to its 4 row skipping (RS) versions. The version RSi refers to the version in which the rows in the i-th outlink column were skipped.

since we saw in Section 6.1 that approximately 40% of the hosts have no outlinks. Similarly, corrupting the second column affects approximately 40% of the hosts since 40% of them have no outlinks and 20% of them have exactly one outlink (see Fig. 3(b)). Corrupting the columns with higher indexes affects significantly smaller number of hosts due to the power law distribution of out-degrees. We chose to skip only the first four columns to simulate significant anomalies on the web graphs.

We then calculated the similarities $sim(GXa, GXb_{RSi})$, $i = 1..4$ of the corrupted web graphs to GXa. We present the results only for the pair $G29a$-$G29b$ since the results of other pairs were similar. Fig. 7 shows the ratios of the computed similarities with the similarity $sim(GXa, GXb)$ of the original web graphs of our pair. The x-axis of Fig. 7 shows the percentage of affected edges of GXb in each case. There are five plots in the plot, one for each algorithm. Every plot has four points that correspond to the similarity of GXa to each of the four GXb_{RSi} graphs.

We see that Vector Similarity, Sequence Similarity, and Signature Similarity (the bottom three plots) are more sensitive to this anomaly. The similarities computed by these algorithms for all corrupted graphs with GXa give similarity scores that are below the threshold defined in Section 6.2. Especially the scores of Sequence Similarity and Signature Similarity are far below the respective thresholds in cases where one of the first three columns is skipped. Hence, although all these three algorithms detect successfully this type of anomaly, the last two have greater confidence intervals and consequently lower possibility of yielding false positive alerts for the most significant anomalies.

Vertex/Edge Overlap has a gradually decreasing slope and gives similarity scores that are close (either below or above) to the defined threshold except from $sim(GXa, GXb_{RS1})$. Hence, this algorithm may detect this anomaly only in cases where a big number of edges is affected. However, even in these cases the similarity scores are close to the threshold and the detection of the anomaly is doubtful. Finally, Vertex Ranking is almost insensitive in this type of anomaly as it always returns similarity values above its threshold.

The results confirm that Vector Similarity, Sequence Similarity, and Signature Similarity are sensitive to topological changes. In case of Sequence Similarity, topological changes lead the walk algorithm to different hops while traversing the graph. In cases of the other two algorithms, each edge is assigned weights that are explicitly affected by the significance of its source vertex, which in the case of PageRank as the quality score is also affected by the topology of the graph. Finally, both Vertex/Edge Overlap and Vertex Ranking fail to detect the anomaly, indicating their lack of sensitivity to the topological changes especially when the anomaly results in a small drop in the vertex/edge overlap.

In summary, Sequence Similarity outperforms the other methods in detecting row skipping anomalies; however, Signature Similarity and Vector Similarity are also successful.

7 Other Related Work

As discussed in Section 5, graph comparison and graph similarity has received a lot of attention in different domains. The book [6] provides a good overview.

Some of the existing similarity schemes were discussed in Section 5 as the basis for our own proposed algorithms. Beyond those schemes, there are other schemes that we think are either not applicable to our problem or whose solutions do not scale to the size of the web. Here we summarize two of these well-known approaches.

Graph isomorphism. This approach, which is discussed at length in [6], is based on the rule that "two graphs are similar if they are the same (up to labels) or one is contained in the other". In other words, this approach tries to find out if two graphs are isomorphic to each other (graph isomorphism problem) or one is isomorphic to a subgraph of the other (subgraph isomorphism problem). The former problem is believed to be NP-Complete, and the latter is provably NP-Complete. This approach is not applicable to web graph similarity as isomorphism is too strong a requirement for web graph comparison. Moreover, as the current exact algorithms for this approach all take exponential time, they cannot be applied to real web graphs at all.

Neighborhood similarity. This approach is built upon the notion of vertex similarity, which in turn is based on the rule that "two vertices are similar if their neighbors are similar". This rule appears to have been independently proposed by [2, 16, 17]. These references proposed different ways of computing similarity scores for pairs of vertices. For some of these references, it is also possible to compute a similarity score for graphs once a similarity score for vertex pairs is known. The proposed algorithms have all been iterative, usually involving a fixpoint computation. They are also expensive as their running times are at least quadratic in the number of vertices of one or both graphs. Thus we believe they are not appropriate for our problem either.

8 Conclusions

Web graphs are at the heart of search engines. Many computers (both internal to the search organization and external to it) are involved in its computation. They are very large and stored on many computers. Because of these complexities many things can go wrong and such anomalies are difficult to detect.

With experimental analysis on real web graphs, we showed that it is feasible to detect such anomalies through similarity computations among consecutive web graphs. In particular, we explored a variety of similarity schemes to be used in detecting when a web graph has changed "more than expected," and can thus be used to detect anomalies. To test the effectiveness of the different schemes, we simulated multiple anomaly cases from three different and realistic anomaly families and experimentally showed which schemes were effective in detecting the anomalies. Table 1 summarizes the results of our experiments. Each row of the table refers to a different similarity scheme and each column to a different family of anomalies.

Of the five schemes explored, Signature Similarity was the best for detecting what we consider significant anomalies, while not yielding false positives when the changes were not significant. Signature Similarity also appears to be good for monitoring the evolution of web crawls. Vector Similarity gave

Scheme \ Anomaly	Row Skipping	Missing Connected	Missing Random
Vertex Ranking	bad	very good	bad
Vector Similarity	good	very good	good
Vertex/Edge Overlap	bad	very good	good
Sequence Similarity	very good	good	bad
Signature Similarity	very good	very good	very good

Table 1: Effectiveness of similarity schemes in detecting anomalies. Signature Similarity appears to be the most effective among these schemes.

also promising results in all the studied cases, while the other three schemes proved to be unsuccessful at detecting anomalies of certain types.

The common feature of the two successful schemes is the use of appropriately weighted web graph features. The proposed algorithms are independent from the weighting schema that is used and, hence, we believe that they can be effective in anomalies that are not studied here. Our future work will include research on feature selection and weighting schemas for the detection of different types of anomalies in the web graph component of search engine.

References

[1] L. Becchetti and C. Castillo. The distribution of PageRank follows a power-law only for particular values of the damping factor. In *Proc. of Int. World Wide Web Conf. (WWW)*, pages 941–942, New York, NY, USA, 2006. ACM.

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

[3] P. Boncz and M. Kersten. MIL primitives for querying a fragmented world. *The VLDB J.*, 8(2):101–119, Mar 1999.

[4] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. Link analysis ranking: Algorithms, theory, and experiments. *ACM Trans. on Internet Tech.*, 5(1):231–297, Feb 2005.

[5] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proc. of Int. World Wide Web Conf. (WWW)*, pages 393–404, Apr 1997.

[6] H. Bunke, P. J. Dickinson, M. Kraetzl, and W. D. Wallis. *A Graph-Theoretic Approach to Enterprise Network Dynamics*. Birkhauser, Boston, MA, 2007.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX Symp. on Operating Syst. Design and Impl. (OSDI)*, pages 205–218. ACM, Nov 2006.

[8] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of Symp. on Theory of Comput. (STOC)*, pages 380–388. ACM, May 2002.

[9] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of VLDB Conf.*, pages 200–209, 2000.

[10] D. Dhyani, W. K. Ng, and S. S. Bhowmick. A survey of web metrics. *ACM Computing Surveys*, 34(4):469–503, Dec 2002.

[11] N. Eiron, K. S. McCurley, and J. A. Tomlin. Ranking the web frontier. In *Proc. of Int. World Wide Web Conf. (WWW)*, pages 309–318. ACM, 2004.

[12] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[13] J. Fan and Q. Yao. *Nonlinear Time Series: Nonparametric and Parametric Methods*. Springer Series in Statistics. Springer-Verlag, New York, NY, 2nd edition, 2005.

[14] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. of VLDB Conf.*, pages 721–732. ACM, Aug 2005.

[15] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proc. of Int. Conf. on Information Retrieval (SIGIR)*, pages 284–291. ACM, Aug 2006.

[16] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *Proc. of Knowledge Discovery and Data Mining Conf. (KDD)*, pages 538–543. ACM, Aug 2002.

[17] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its applications to schema matching. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 117–128. ACM, Apr 2002.

[18] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In *Proc. of Int. World Wide Web Conf. (WWW)*, pages 1–12, New York, NY, USA, 2004. ACM.

[19] A. Papadopoulos and Y. Manolopoulos. Structure-based similarity search with graph histograms. In *Proc. of DEXA/IWOSS Int. Wrkshp on Similarity Search*, pages 174–178. IEEE, 1999.

[20] A. Robles-Kelley and E. R. Hancock. Graph edit distance from spectral seriation. *IEEE T. on Pattern Analysis and Machine Intelligence*, 27(3):365–378, Mar 2005.

[21] T. Seidl. References for graph similarity. URL: http://www.dbs.informatik.uni-muenchen.de/~seidl/graphs/, Aug 2007.

[22] G. R. Xue, Q. Yang, H. J. Zeng, Y. Yu, and Z. Chen. Exploiting the hierarchical structure for link analysis. In *Proc. of Conf. on Information Retrieval (SIGIR)*, pages 186–193, New York, NY, USA, 2005. ACM.

[23] L. Zager and G. Verghese. Graph similarity. URL: http://lees.mit.edu/lees/presentations/LauraZager.pdf, Mar 2005.

[24] P. Zhu and R. C. Wilson. A study of graph spectra for comparing graphs. In *Proc. of British Machine Vision Conf. (MBVC)*, Sep 2005.