The h-Index of a Graph and Its Application to Dynamic Subgraph Statistics

David Eppstein[1] and Emma S. Spiro[2]

[1] Computer Science Department, University of California, Irvine
[2] Department of Sociology, University of California, Irvine

Abstract. We describe a data structure that maintains the number of triangles in a dynamic undirected graph, subject to insertions and deletions of edges and of degree-zero vertices. More generally it can be used to maintain the number of copies of each possible three-vertex subgraph in time $O(h)$ per update, where h is the *h-index* of the graph, the maximum number such that the graph contains h vertices of degree at least h. We also show how to maintain the h-index itself, and a collection of h high-degree vertices in the graph, in constant time per update. Our data structure has applications in social network analysis using the exponential random graph model (ERGM); its bound of $O(h)$ time per edge is never worse than the $\Theta(\sqrt{m})$ time per edge necessary to list all triangles in a static graph, and is strictly better for graphs obeying a power law degree distribution. In order to better understand the behavior of the h-index statistic and its implications for the performance of our algorithms, we also study the behavior of the h-index on a set of 136 real-world networks.

1 Introduction

The *exponential random graph model* (ERGM, or p^* model) [18, 35, 30] is a general technique for assigning probabilities to graphs that can be used both to generate simulated data for social network analysis and to perform probabilistic reasoning on real-world data. In this model, one fixes the vertex set of a graph, identifies certain *features* f_i in graphs on that vertex set, determines a *weight* w_i for each feature, and sets the probability of each graph G to be proportional to an exponential function of the sum of its features' weights, divided by a normalizing constant Z:

$$\Pr(G) = \frac{\exp \sum_{f_i \in G} w_i}{Z}.$$

Z is found by summing over all graphs on that vertex set:

$$Z = \sum_G \exp \sum_{f_i \in G} w_i.$$

For instance, if each potential edge is considered to be a feature and all edges have weight $\ln \frac{p}{1-p}$, the normalizing constant Z will be $(1-p)^{-n(n-1)/2}$, and the probability of any particular m-edge graph will be $p^m(1-p)^{n(n-1)/2-m}$, giving rise to the familiar Erdős-Rényi $G(n,p)$ model. However, the ERG model is much more general than

F. Dehne et al. (Eds.): WADS 2009, LNCS 5664, pp. 278–289, 2009.

the Erdős-Rényi model: for instance, an ERGM in which the features are whole graphs can represent arbitrary probabilities. The generality of this model, and its ability to define probability spaces lacking the independence properties of the simpler Erdős-Rényi model, make it difficult to analyze analytically. Instead, in order to generate graphs in an ERG model or to perform other forms of probabilistic reasoning with the model, one typically uses a Markov Chain Monte Carlo method [31] in which one performs a large sequence of small changes to sample graphs, updates after each change the counts of the number of features of each type and the sum of the weights of each feature, and uses the updated values to determine whether to accept or reject each change. Because this method must evaluate large numbers of graphs, it is important to develop very efficient algorithms for identifying the features that are present in each graph.

Typical features used in these models take the form of small subgraphs: *stars* of several edges with a common vertex (used to represent constraints on the degree distribution of the resulting graphs), *triangles* (used in the triad model [19], an important predecessor of ERG models, to represent the likelihood that friends-of-friends are friends of each other), and more complicated subgraphs used to control the tendencies of simpler models to generate unrealistically extremal graphs [32]. Using highly local features of this type is important for reasons of computational efficiency, matches well the type of data that can be obtained for real-world social networks, and is well motivated by the local processes believed to underly many types of social network. Thus, ERGM simulation leads naturally to problems of *subgraph isomorphism*, listing or counting all copies of a given small subgraph in a larger graph.

There has been much past algorithmic work on subgraph isomorphism problems. It is known, for instance, that an n-vertex graph with m edges may have $\Theta(m^{3/2})$ triangles and four-cycles, and all triangles and four-cycles can be found in time $O(m^{3/2})$ [22,6]. All cycles of length up to seven can be counted rather than listed in time of $O(n^{\omega})$ [3] where $\omega \approx 2.376$ is the exponent from the asymptotically fastest known matrix multiplication algorithms [7]; this improves on the previous $O(m^{3/2})$ bounds for dense graphs. Fast matrix multiplication has also been used for more general problems of finding and counting small cliques in graphs and hypergraphs [10,24,26,34,36]. In planar graphs, or more generally graphs of bounded local treewidth, the number of copies of any fixed subgraph may be found in linear time [13, 14], even though this number may be a large polynomial of the graph size [11]. Approximation algorithms for subgraph isomorphism counting problems based on random sampling have also been studied, with motivating applications in bioinformatics [9,23,29]. However, much of this subgraph isomorphism research makes overly restrictive assumptions about the graphs that are allowed as input, runs too slowly for the ERGM application, depends on impractically complicated matrix multiplication algorithms, or does not capture the precise subgraph counts needed to accurately perform Markov Chain Monte Carlo simulations.

Markov Chain Monte Carlo methods for ERGM-based reasoning process a sequence of graphs each differing by a small change from a previous graph, so it is natural to seek additional efficiency by applying *dynamic graph algorithms* [15,17,33], data structures to efficiently maintain properties of a graph subject to vertex and edge insertions and deletions. However, past research on dynamic graph algorithms has focused on problems of connectivity, planarity, and shortest paths, and not on finding the features needed in ERGM calculations. In this paper, we apply dynamic graph algorithms to subgraph isomorphism problems important in ERGM feature identification. To our knowledge, this is the first work on dynamic algorithms for subgraph isomorphism.

A key ingredient in our algorithms is the h-index, a number introduced by Hirsch [21] as a way of balancing prolixity and impact in measuring the academic achievements of individual researchers. Although problematic in this application [1], the h-index can be defined and studied mathematically, in graph-theoretic terms, and provides a convenient measure of the uniformity of distribution of edges in a graph. Specifically, for a researcher, one may define a bipartite graph in which the vertices on one side of the bipartition represent the researcher's papers, the vertices on the other side represent others' papers, and edges correspond to citations by others of the researcher's papers. The h-index of the researcher is the maximum number h such that at least h vertices on the researcher's side of the bipartition each have degree at least h. We generalize this to arbitrary graphs, and define the h-index of any graph to be the maximum h such that the graph contains h vertices of degree at least h. Intuitively, an algorithm whose running time is bounded by a function of h is capable of tolerating arbitrarily many low-degree vertices without slowdown, and is only mildly affected by the presence of a small number of very high degree vertices; its running time depends primarily on the numbers of intermediate-degree vertices. As we describe in more detail in Section 7, the h-index of any graph with m edges and n vertices is sandwiched between m/n and $\sqrt{2m}$, so it is sublinear whenever the graph is not dense, and the worst-case graphs for these bounds have an unusual degree distribution that is unlikely to arise in practice.

Our main result is that we may maintain a dynamic graph, subject to edge insertions, edge deletions, and insertions or deletions of isolated vertices, and maintain the number of triangles in the graph, in time $O(h)$ per update where h is the h-index of the graph at the time of the update. This compares favorably with the time bound of $\Theta(m^{3/2})$ necessary to list all triangles in a static graph. In the same $O(h)$ time bound per update we may more generally maintain the numbers of three-vertex induced subgraphs of each possible type, and in constant time per update we may maintain the h-index itself. Our algorithms are randomized, and our analysis of them uses amortized analysis to bound their expected times on worst-case input sequences. Our use of randomization is limited, however, to the use of hash tables to store and retrieve data associated with keys in $O(1)$ expected time per access. By using either direct addressing or deterministic integer searching data structures instead of hash tables we may avoid the use of randomness at an expense of either increased space complexity or an additional factor of $O(\log\log n)$ in time complexity; we omit the details.

We also study the behavior of the h-index, both on scale-free graph models and on a set of real-world graphs used in social network analysis. We show that for scale-free graphs, the h-index scales as a power of n, less than its square root, while in the real-world graphs we studied the scaling exponent appears to have a bimodal distribution.

2 Dynamic h-Indexes of Integer Functions

We begin by describing a data structure for the following problem, which generalizes that of maintaining h-indexes of dynamic graphs. We are given a set S, and a function

f from S to the non-negative integers, both of which may vary discretely through a sequence of updates: we may insert or delete elements of S (with arbitrary function values for the inserted elements), and we may make arbitrary changes to the function value of any element of S. As we do so, we wish to maintain a set H such that, for every $x \in H$, $f(x) \geq |H|$, with H as large as possible with this property. We call $|H|$ the *h-index* of S and f, and we call the partition of S into the two subsets $(H, S \setminus H)$ an *h-partition* of S and f.

To do so, we maintain the following data structures:

- A dictionary F mapping each $x \in S$ to its value under f: $F[x] = f(x)$.
- The set H (stored as a dictionary mapping members of H to an arbitrary value).
- The set $B = \{x \in H \mid f(x) = |H|\}$.
- A dictionary C mapping each non-negative integer i to the set $\{x \in S \setminus B \mid f(x) = i\}$. We only store these sets when they are non-empty, so the situation that there is no x with $f(x) = i$ can be detected by the absense of i among the keys of C.

To insert an element x into our structure, we first set $F[x] = f(x)$, and add x to $C[f(x)]$ (or add a new set $\{x\}$ at $C[f(x)]$ if there is no existing entry for $f(x)$ in C). Then, we test whether $f(x) > |H|$. If not, the h-index does not change, and the insertion operation is complete. But if $f(x) > |H|$, we must include x into H. If B is nonempty, we choose an arbitrary $y \in B$, remove y from B and from H, and add y to $C[|H|]$ (or create a new set $\{y\}$ if there is no entry for $|H|$ in C). Finally, if $f(x) > |H|$ and B is empty, the insertion causes the h-index ($|H|$) to increase by one. In this case, we test whether there is an entry for the new value of $|H|$ in C. If so, we set B to equal the identity of the set in $C[|H|]$ and delete the entry for $|H|$ in C; otherwise, we set B to the empty set.

To remove x from our structure, we remove its entry from F and we remove it from B (if it belongs there) or from the appropriate set in $C[f(x)]$ otherwise. If x did not belong to H, the h-index does not change, and the deletion operation is complete. Otherwise, let h be the value of $|H|$ before removing x. We remove x from H, and attempt to restore the lost item from H by moving an element from $C[h]$ to B (deleting $C[h]$ if this operation causes it to become empty). But if C has no entry for h, the h-index decreases; in this case we store the identity of set B into $C[h]$, and set B to be the empty set.

Changing the value of $f(x)$ may be accomplished by deleting x and then reinserting it, with some care so that we do not update H if x was already in H and both the old and new values of $f(x)$ are at least equal to $|H|$.

Theorem 1. *The data structure described above maintains the h-index of S and f, and an h-partition of S and f, in constant time plus a constant number of dictionary operations per update.*

We defer the proof to the full version of the paper [16].

3 Gradual Approximate h-Partitions

Although the vector h-index data structure of the previous section allows us to maintain the h-index of a dynamic graph very efficiently, it has a property that would be undesirable were we to use it directly as part of our later dynamic graph data structures:

the h-partition $(H, S \setminus H)$ changes too frequently. Changes to the set H will turn out to be such an expensive operation that we only wish them to happen, on average, $O(1/h)$ times per update. In order to achieve such a small amount of change to H, we need to restrict the set of updates that are allowed: now, rather than arbitrary changes to f, we only allow it to be incremented or decremented by a single unit, and we only allow an element x to be inserted or deleted when $f(x) = 0$. We now describe a modification of the H-partition data structure that has this property of changing more gradually for this restricted class of updates.

Specifically, along with all of the structures of the H-partition, we maintain a set $P \subset H$ describing a partition $(P, S \setminus P)$. When an element of x is removed from H, we remove it from P as well, to maintain the invariant that $P \subset H$. However, we only add an element x to P when an update (an increment of $f(x)$ or decrement of $f(y)$ for some other element y) causes $f(x)$ to become greater than or equal to $2|H|$. The elements to be added to P on each update may be found by maintaining a dictionary, parallel to C, that maps each integer i to the set $\{x \in H \setminus P \mid f(x) = i\}$.

Theorem 2. *Let σ denote a sequence of operations to the data structure described above, starting from an empty data structure. Let h_t denote the value of h after t operations, and let $q = \sum_i 1/h_i$. Then the data structure undergoes $O(q)$ additions and removals of an element to or from P.*

We defer the proof to the full version of the paper [16]. For our later application of this technique as a subroutine in our triangle-finding data structure, we will need a more local analysis. We may divide a sequence of updates into *epochs*, as follows: each epoch begins when the h-index reaches a value that differs from the value at the beginning of the previous epoch by a factor of two or more. Then, as we show in the full version, an epoch with h as its initial h-index lasts for at least $\Omega(h^2)$ steps. Due to this length, we may assign a full unit of credit to each member of P at the start of each epoch, without changing the asymptotic behavior of the total number of credits assigned over the course of the algorithm. With this modification, it follows from the same analysis as above that, within an epoch of s steps, with an h-index of h at the start of the epoch, there are $O(s/h)$ changes to P.

4 Counting Triangles

We are now ready to describe our data structure for maintaining the number of triangles in a dynamic graph. It consists of the following information:

- A count of the number of triangles in the current graph
- A set E of the edges in the graph, indexed by the pair of endpoints of the edge, allowing constant-time tests for whether a given pair of endpoints are linked by an edge.
- A partition of the graph vertices into two sets H and $V \setminus H$ as maintained by the data structure from Section 3.
- A dictionary P mapping each pair of vertices u, v to a number $P[u, v]$, the number of two-edge paths from u to v via a vertex of $V \setminus H$. We only maintain nonzero values for this number in P; if there is no entry in P for the pair u, v then there exist no two-edge paths via $V \setminus H$ that connect u to v.

Theorem 3. *The data structure described above requires space $O(mh)$ and may be maintained in $O(h)$ randomized amortized time per operation, where h is the h-index of the graph at the time of the operation.*

Proof. Insertion and deletion of vertices with no incident edges requires no change to most of these data structures, so we concentrate our description on the edge insertion and deletion operations.

To update the count of triangles, we need to know the number of triangles uvw involving the edge uv that is being deleted or inserted. Triangles in which the third vertex w belongs to H may be found in time $O(h)$ by testing all members of H, using the data structure for E to test in constant time per member whether it forms a triangle. Triangles in which the third vertex w does not belong to H may be counted in time $O(1)$ by a single lookup in P.

The data structure for E may be updated in constant time per operation, and the partition into H and $V \setminus H$ may be maintained as described in the previous sections in constant time per operation. Thus, it remains to describe how to update P. If we are inserting an edge uv, and u does not belong to H, it has at most $2h$ neighbors; we examine all other neighbors w of u and for each such neighbor increment the counter in $P[v,w]$ (or create a new entry in $P[v,w]$ with a count of 1 if no such entry already exists). Similarly if v does not belong to H we examine all other neighbors w of v and for each such neighbor increment $P[u,w]$. If we are deleting an edge, we similarly decrement the counters or remove the entry for a counter if decrementing it would leave a zero value. Each update involves incrementing or decrementing $O(h)$ counters and therefore may be implemented in $O(h)$ time.

Finally, a change to the graph may lead to a change in H, which must be reflected in P. If a vertex v is moved from H to $V \setminus H$, we examine all pairs u,w of neighbors of v and increment the corresponding counts in $P[u,w]$, and if a vertex v is moved from $V \setminus H$ to H we examine all pairs u,w of neighbors of v and decrement the corresponding counts in $P[u,w]$. This step takes time $O(h^2)$, because v has $O(h)$ neighbors when it is moved in either direction, but as per the analysis in Section 3 it is performed an average of $O(1/h)$ times per operation, so the amortized time for updates of this type, per change to the input graph, is $O(h)$.

The space for the data structure is $O(m)$ for E, $O(n)$ for the data structure that maintains H, and $O(mh)$ for P because each edge of the graph belongs to $O(h)$ two-edge paths through low-degree vertices. □

5 Subgraph Multiplicity

Although the data structure of Theorem 3 only counts the number of triangles in a graph, it is possible to use it to count the number of three-vertex subgraphs of all types, or the number of induced three-vertex subgraphs of all types. In what follows we let $p_i = p_i(G)$ denote the number of paths of length i in G, and we let $c_i = c_i(G)$ denote the number of cycles of length i in G.

The set of all edges in a graph G among a subset of three vertices $\{u,v,w\}$ determine one of four possible induced subgraphs: an independent set with no edges, a graph with a single edge, a two-star consisting of two edges, or a triangle. Let g_0, g_1, g_2, and g_3

denote the numbers of three-vertex subgraphs of each of these types, where g_i counts the three-vertex induced subgraphs that have i edges.

Observe that it is trivial to maintain for a dynamic graph, in constant time per operation, the three quantities n, m, and p_2, where n denotes the number of vertices of the graph, m denotes the number of edges, and p_2 denotes the number of two-edge paths that can be formed from the edges of the graph. Each change to the graph increments or decrements n or m. Additionally, adding an edge uv to a graph where u and v already have d_u and d_v incident edges respectively increases p_2 by $d_u + d_v$, while removing an edge uv decreases p_2 by $d_u + d_v - 2$. Letting c_3 denote the number of triangles in the graph as maintained by Theorem 3, the quantities described above satisfy the matrix equation

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{bmatrix} = \begin{bmatrix} n(n-1)(n-2)/6 \\ m(n-2) \\ p_2 \\ c_3 \end{bmatrix}.$$

Each row of the matrix corresponds to a single linear equation in the g_i values. The equation from the first row, $g_0 + g_1 + g_2 + g_3 = \binom{n}{3}$, can be interpreted as stating that all triples of vertices form one graph of one of these types. The equation from the second row, $g_1 + 2g_2 + 3g_3 = m(n-2)$, is a form of double counting where the number of edges in all three-vertex subgraphs is added up on the left hand side by subgraph type and on the right hand side by counting the number of edges (m) and the number of triples each edge participates in $(n-2)$. The third row's equation, $g_2 + 3g_3 = p_2$, similarly counts incidences between two-edge paths and triples in two ways, and the fourth equation $g_3 = c_3$ follows since each three vertices that are connected in a triangle cannot form any other induced subgraph than a triangle itself.

By inverting the matrix we may reconstruct the g values:

$$\begin{aligned} g_3 &= c_3 \\ g_2 &= p_2 - 3g_3 \\ g_1 &= m(n-2) - (2g_2 + 3g_3) \\ g_0 &= \binom{n}{3} - (g_1 + g_2 + g_3). \end{aligned}$$

Thus, we may maintain each number of induced subgraphs g_i in the same asymptotic time per update as we maintain the number of triangles in our dynamic graph. The numbers of subgraphs of different types that are not necessarily induced are even easier to recover: the number of three-vertex subgraphs with i edges is given by the ith entry of the vector on the right hand side of the matrix equation.

As we detail in the full version of the paper [16], it is also possible to maintain efficiently the numbers of star subgraphs of a dynamic graph, and the number of four-vertex paths in a dynamic graph.

6 Weighted Edges and Colored Vertices

It is possible to generalize our triangle counting method to problems of weighted triangle counting: we assign each edge uv of the graph a weight w_{uv}, define the weight of a triangle to be the product of the weights of its edges, and maintain the total weight of all triangles. For instance, if $0 \leq w_{uv} \leq 1$ and each edge is present in a subgraph with probability w_{uv}, then the total weight gives the expected number of triangles in that subgraph.

Theorem 4. *The total weight of all triangles in a weighted dynamic graph, as described above, may be maintained in time $O(h)$ per update.*

Proof. We modify the structure $P[u,v]$ maintained by our triangle-finding data structure, so that it stores the weight of all two-edge paths from u to v. Each update of an edge uv in our structure involves a set of individual triangles uvx involving vertices $x \in H$ (whose weight is easily calculated) together with the triangles formed by paths counted in $P[u,v]$ (whose total weight is $P[u,v]w_{uv}$). The same time analysis from Theorem 3 holds for this modified data structure. □

For social networking ERGM applications, an alternative generalization may be appropriate. Suppose that the vertices of the given dynamic graph are colored; we wish to maintain the number of triangles with each possible combination of colors. For instance, in graphs representing sexual contacts [25], edges between individuals of the same sex may be less frequent than edges between individuals of opposite sexes; one may model this in an ERGM by assigning the vertices two different colors according to whether they represent male or female individuals and using feature weights that depend on the colors of the vertices in the features. As we now show, problems of counting colored triangles scale well with the number of different groups into which the vertices of the graph are classified.

Theorem 5. *Let G be a dynamic graph in which each vertex is assigned one of k different colors. Then we may maintain the numbers of triangles in G with each possible combination of colors, in time $O(h+k)$ per update.*

Proof. We modify the structure $P[u,v]$ stored by our triangle-finding data structure, to store a vector of k numbers: the ith entry in this vector records the number of two-edge paths from u to v through a low-degree vertex with color i. Each update of an edge uv in our structure involves a set of individual triangles uvx involving vertices $x \in H$ (whose colors are easily observed) together with the triangles formed by paths counted in $P[u,v]$ (with k different possible colorings, recorded by the entries in the vector $P[u,v]$). Thus, the part of the update operation in which we compute the numbers of triangles for which the third vertex has low degree, by looking up u and v in P, takes time $O(k)$ instead of $O(1)$. The same time analysis from Theorem 3 holds for all other aspects of this modified data structure. □

Both the weighting and coloring generalizations may be combined with each other without loss of efficiency.

7 How Small Is the h-Index of Typical Graphs?

It is straightforward to identify the graphs with extremal values of the h-index. A split graph in which an h-vertex clique is augmented by adding $n-h$ vertices, each connected only to the vertices in the clique, has n vertices and $m = h(n-1)$ edges, achieving an h-index of $m/(n-1)$. This is the minimum possible among any graph with n vertices and m edges: any other graph may be transformed into a split graph of this type, while increasing its number of edges and not decreasing h, by finding an h-partition $(H, V \setminus H)$ and repeatedly replacing edges that do not have an endpoint in H by edges that do have such an endpoint. The graph with the largest h-index is a clique with m edges together with enough isolated vertices to fill out the total to n; its h-index is $\sqrt{2m}(1+o(1))$. Thus, for sparse graphs in which the numbers of edges and vertices are proportional to each other, the h-index may be as small as $O(1)$ or as large as $\Omega(\sqrt{n})$. At which end of this spectrum can we expect to find the graphs arising in social network analysis?

One answer can be provided by fitting mathematical models of the *degree distribution*, the relation between the number of incident edges at a vertex and the number of vertices with that many edges, to social networks. For many large real-world graphs, observers have reported *power laws* in which the number of vertices with degree d is proportional to $nd^{-\gamma}$ for some constant $\gamma > 1$; a network with this property is called *scale-free* [2,25,27,28]. Typically, γ lies in or near the interval $2 \leq \gamma \leq 3$ although more extreme values are possible. The h-index of these graphs may be found by solving for the h such that $h = nh^{-\gamma}$; that is, $h = \Theta(n^{1/(1+\gamma)})$. For any $\gamma > 1$ this is an asymptotic improvement on the worst-case $O(\sqrt{n})$ bound for graphs without power-law degree distributions. For instance, for $\gamma = 2$ this would give a bound of $h = O(n^{1/3})$ while for $\gamma = 3$ it would give $h = O(n^{1/4})$. That is, by depending on the h-index as it does, our algorithm is capable of taking advantage of the extra structure inherent in scale-free graphs to run more quickly for them than it does in the general case.

To further explore h-index behavior in real-world networks, we computed the h-index for a collection of 136 network data sets typical of those used in social network analysis. These data sets were drawn from a variety of sources traditionally viewed as common repositories for such data. The majority of our data sets were from the well known Pajek datasets [4]. Pajek is a program used for the analysis and visualization of large networks. The collection of data available with the Pajek software includes citation networks, food-webs, friendship network, etc. In addition to the Pajek data sets, we included network data sets from UCINET [5]. Another software package developed for network analysis, UCINET includes a corpus of data sets that are more traditional in the social sciences. Many of these data sets represent friendship or communication relations; UCINET also includes various social networks for non-human animals. We also used network data included as part of the statnet software suite [20], statistical modeling software in R. statnet includes ERGM functionality, making it a good example for data used specifically in the context of ERG models. Finally, we included data available on the UCI Network Data Repository [8], including some larger networks such as the WWW, blog networks, and other online social networks. By using this data we hope to understand how the h-index scales in real-world networks.

Details of the statistics for these networks are presented in the full version of the paper [16]; a summary of the statistics for network size and h-index are in Table 1, below. For this sample of 136 real-world networks, the h-index ranges from 2 to 116. The row of summary statistics for $\log h / \log n$ suggests that, for many networks, h scales as a sublinear power of n. The one case with an h-index of 116 represents the ties among

Table 1. Summary statistics for real-world network data

	min.	median	mean	max.
network size (n)	10	67	535.3	10616
h-index (h)	2	12	19.08	116
$\log n$	2.303	4.204	4.589	9.270
$\log h$	0.6931	2.4849	2.6150	4.7536
$\log h / \log n$	0.2014	0.6166	0.6006	1.0000

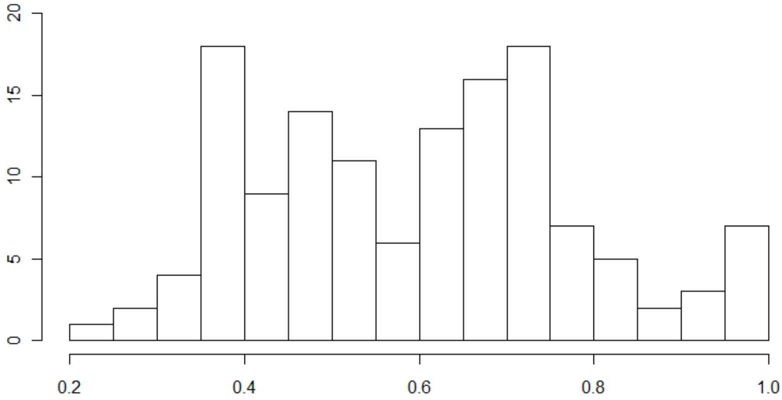

Fig. 1. A frequency histogram for $\log h / \log n$

Slovenian magazines and journals between 1999 and 2000. The vertices of this network represent journals, and undirected edges between journals have an edge weight that represents the number of shared readers of both journals; this network also includes self-loops describing the number of all readers that read this journal. Thus, this is a dense graph, more appropriately handled using statistics involving the edge weights than with combinatorial techniques involving the existence or nonexistence of triangles. However, this is the only network from our dataset with an h-index in the hundreds. Even with significantly larger networks, the h-index appears to scale sublinearly in most cases.

A histogram of the h-index data in Figure 1 clearly shows a bimodal distribution. Additionally, as the second peak of the bimodal distribution corresponds to a scaling exponent greater than 0.5, the graphs corresponding to that peak do not match the predictions of the scale-free model. However we were unable to discern a pattern to the types of networks with smaller or larger h-indices, and do not speculate on the reasons for this bimodality. We look more deeply at the scaling of the h-index using standard regression techniques in the full version of the paper [16].

8 Discussion

We have defined an interesting new graph invariant, the h-index, presented efficient dynamic graph algorithms for maintaining the h-index and, based on them, for maintaining the set of triangles in a graph, and studied the scaling behavior of the h-index both on theoretical scale-free graph models and on real-world network data.

There are many directions for future work. For sparse graphs, the h-index may be larger than the *arboricity*, a graph invariant used in static subgraph isomorphism [6,12]; can we speed up our dynamic algorithms to run more quickly on graphs of bounded arboricity? We handle undirected graphs but the directed case is also of interest. We would like to find efficient data structures to count larger subgraphs such as 4-cycles, 4-cliques, and claws; dynamic algorithms for these problems are likely to be slower than our triangle-finding algorithms but may still provide speedups over static algorithms. Another network statistic related to triangle counting is the clustering coefficient of a graph; can we maintain it efficiently? Additionally, there is an opportunity for additional work in implementing our data structures and testing their efficiency in practice.

Acknowledgements. This work was supported in part by NSF grant 0830403 and by the Office of Naval Research under grant N00014-08-1-1015.

References

1. Adler, R., Ewing, J., Taylor, P.: Citation Statistics: A report from the International Mathematical Union (IMU) in cooperation with the International Council of Industrial and Applied Mathematics (ICIAM) and the Institute of Mathematical Statistics. In: Joint Committee on Quantitative Assessment of Research (2008)
2. Albert, R., Jeong, H., Barabasi, A.-L.: The diameter of the world wide web. Nature 401, 130–131 (1999)
3. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
4. Batagelj, V., Mrvar, A.: Pajek datasets (2006), `http://vlado.fmf.uni-lj.si/pub/networks/data/`
5. Borgatti, S.P., Everett, M.G., Freeman, L.C.: UCINet 6 for Windows: Software for social network analysis. Analytic Technologies, Harvard, MA (2002)
6. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM J. Comput. 14(1), 210–223 (1985)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3), 251–280 (1990)
8. DuBois, C.L., Smyth, P.: UCI Network Data Repository (2008), `http://networkdata.ics.uci.edu`
9. Duke, R.A., Lefmann, H., Rödl, V.: A fast approximation algorithm for computing the frequencies of subgraphs in a given graph. SIAM J. Comput. 24(3), 598–620 (1995)
10. Eisenbrand, F., Grandoni, F.: On the complexity of fixed parameter clique and dominating set. Theoretical Computer Science 326(1–3), 57–67 (2004)
11. Eppstein, D.: Connectivity, graph minors, and subgraph multiplicity. Journal of Graph Theory 17, 409–416 (1993)
12. Eppstein, D.: Arboricity and bipartite subgraph listing algorithms. Information Processing Letters 51(4), 207–211 (1994)
13. Eppstein, D.: Subgraph isomorphism in planar graphs and related problems. Journal of Graph Algorithms & Applications 3(3), 1–27 (1999)
14. Eppstein, D.: Diameter and treewidth in minor-closed graph families. Algorithmica 27, 275–291 (2000)

15. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Atallah, M.J. (ed.) Algorithms and Theory of Computation Handbook, ch. 8, CRC Press, Boca Raton (1999)
16. Eppstein, D., Spiro, E.S.: The h-index of a graph and its application to dynamic subgraph statistics. Electronic preprint arxiv:0904.3741 (2009)
17. Feigenbaum, J., Kannan, S.: Dynamic graph algorithms. In: Rosen, K. (ed.) Handbook of Discrete and Combinatorial Mathematics. CRC Press, Boca Raton (2000)
18. Frank, O.: Statistical analysis of change in networks. Statistica Neerlandica 45, 283–293 (1999)
19. Frank, O., Strauss, D.: Markov graphs. J. Amer. Statistical Assoc. 81, 832–842 (1986)
20. Handcock, M.S., Hunter, D., Butts, C.T., Goodreau, S.M., Morris, M.: statnet: An R package for the Statistical Modeling of Social Networks (2003), `http://www.csde.washington.edu/statnet`
21. Hirsch, J.E.: An index to quantify an individual's scientific research output. Proc. National Academy of Sciences 102(46), 16569–16572 (2005)
22. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM J. Comput. 7(4), 413–423 (1978)
23. Kashtan, N., Itzkovitz, S., Milo, R., Alon, U.: Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. Bioinformatics 20(11), 1746–1758 (2004)
24. Kloks, T., Kratsch, D., Müller, H.: Finding and counting small induced subgraphs efficiently. Information Processing Letters 74(3–4), 115–121 (2000)
25. Liljeros, F., Edling, C.R., Amaral, L.A.N., Stanley, H.E., Åberg, Y.: The web of human sexual contacts. Nature 411, 907–908 (2001)
26. Nešetřil, J., Poljak, S.: On the complexity of the subgraph problem. Commentationes Mathematicae Universitatis Carolinae 26(2), 415–419 (1985)
27. Newman, M.E.J.: The structure and function of complex networks. SIAM Review 45, 167–256 (2003)
28. desolla Price, D.J.: Networks of scientific papers. Science 149(3683), 510–515 (1965)
29. Pržulj, N., Corneil, D.G., Jurisica, I.: Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. Bioinformatics 22(8), 974–980 (2006)
30. Robins, G., Morris, M.: Advances in exponential random graph (p^*) models. Social Networks 29(2), 169–172 (2007); Special issue of journal with four additional articles
31. Snijders, T.A.B.: Markov chain Monte Carlo estimation of exponential random graph models. Journal of Social Structure 3(2), 1–40 (2002)
32. Snijders, T.A.B., Pattison, P.E., Robins, G., Handcock, M.S.: New specifications for exponential random graph models. Sociological Methodology 36(1), 99–153 (2006)
33. Thorup, M., Karger, D.R.: Dynamic graph algorithms with applications. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 667–673. Springer, Heidelberg (2000)
34. Vassilevska, V., Williams, R.: Finding, minimizing and counting weighted subgraphs. In: Proc. 41st ACM Symposium on Theory of Computing (2009)
35. Wasserman, S., Pattison, P.E.: Logit models and logistic regression for social networks, I: an introduction to Markov graphs and p^*. Psychometrika 61, 401–425 (1996)
36. Yuster, R.: Finding and counting cliques and independent sets in r-uniform hypergraphs. Information Processing Letters 99(4), 130–134 (2006)