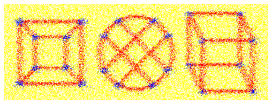
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 16, no. 2, pp. 543–567 (2012)
DOI: 10.7155/jgaa.00273

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

David Eppstein[1] *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.

Submitted:	Accepted:	Final:	Published:
December 2011	August 2012	August 2012	August 2012
	Article type: Regular paper	Communicated by: S. Albers	

This work was supported in part by NSF grants 0830403 and 1217322 and by the Office of Naval Research under grant N00014-08-1-1015. A preliminary version of these results was presented at the Algorithms and Data Structures Symposium (WADS), 2009.

E-mail addresses: eppstein@uci.edu (David Eppstein) espiro@uci.edu (Emma S. Spiro)

1 Introduction

In this paper we study problems of counting the number of copies of a small subgraph that are present in a larger graph undergoing a dynamic sequence of edge addition and edge deletion operations. Although this dynamic subgraph isomorphism problem can be motivated by several different applications, our primary motivation is in the statistical modeling of social networks.

The *exponential random graph model* (ERGM, or p^* model) [19, 33, 38] is a 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 defines the probability of each graph G to be proportional to an exponential function of the sum of its features' weights:

$$\Pr(G) = \frac{1}{Z} \exp \sum_{f_i \in G} w_i.$$

The constant of proportionality Z is found by summing over all graphs on the same vertex set:

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

For example, if each potential edge is considered to be a feature and all edges have weight $\ln \frac{p}{1-p}$, the result is the familiar Erdős–Rényi–Gilbert $G(n, p)$ model of random graphs [21]. However, the ERGM model is much more general: any probability distribution on graphs with a fixed vertex set can be modeled as an ERGM with an appropriate set of features. Because of its generality, the ERGM model is difficult to analyze analytically. Instead, in order to generate graphs in an ERGM model or to perform other forms of probabilistic reasoning with the model such as fitting feature weights to real-world data, one typically uses the Metropolis–Hastings algorithm [34], a version of Markov Chain Monte Carlo sampling that performs a large sequence of small updates to sample graphs, recalculates after each update the sum of feature weights, and uses the updated weight sums to determine whether to accept or reject each update. Because this method must evaluate large numbers of graphs, it is important to develop very efficient algorithms for identifying the features that change after each update. And because only the small set of changed features between two very similar graphs is relevant for determining whether to accept or reject an update, it is important that these algorithms be exact and not approximate.

Typical features used in social network modeling applications of the ERGM framework take the form of small subgraphs. Features in the form of *stars* of several edges with a common vertex may be used to model constraints on the degree distribution of the resulting graphs. *Triangles*, complete three-vertex subgraphs, model the sociological phenomenon that two people who have a friend in common have an increased likelihood of being friends with each other [20]. Other subgraph features may also be used to control the tendencies of simpler

models to generate unrealistically extremal graphs [35]. ERGM simulation with this type of feature leads naturally to algorithmic 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 $O(m^{3/2})$ [7, 24]. 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 [8]; 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 [11,26,29,37,39]. 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 [14, 15], even though this number may be a large polynomial of the graph size [12]. Approximation algorithms for subgraph isomorphism counting problems based on random sampling have also been studied, with motivating applications in bioinformatics [10,25,32]. 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 approximates the subgraph counts rather than calculating them precisely as is needed to accurately perform the Metropolis–Hastings algorithm.

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* [16, 18, 36], 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 [23] 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 own papers, the vertices on the other side represent papers by other people, 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 8, 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.

The rest of this paper is organized as follows. In Section 2 we describe a data structure for maintaining dynamically the h-index of a graph, or more generally of any integer function on a set. As well as the index itself, this data structure maintains a partition of the graph's vertices into a small set of high-degree vertices and a larger set of low-degree vertices. In Section 3 we modify this data structure so that changes to the partition are very infrequent. Our main result, in Section 4, is a data structure for counting the triangles in a dynamic graph; it uses the slowly-changing degree partition of Section 3 as a subroutine. In Section 5 we describe how to use this result to maintain statistics about all three-vertex subgraphs or induced subgraphs in a dynamic graph, in Section 6 we discuss algorithms for maintaining counts of certain subgraphs with more than three vertices, and in Section 7 we discuss variants of these counting problems with weighted edges and colored vertices. In Section 8 we describe the behavior of the h-index on a corpus of over 100 real-world graphs, and we conclude with a section discussing our results.

2 Dynamic h-Indexes of Integer Functions

We begin by describing a data structure for the following problem, which generalizes the problem of maintaining h-indexes of dynamic graphs.

Suppose that we are given a set S, and a function f from S to the non-negative integers; then we define the *h-index* of S and f to be the maximum number h such that there exists a subset $H \subset S$, with $|H| = h$, and which $f(x) \geq h$ for every member of h. We call the partition of S into the two subsets $(H, S \setminus H)$ an *h-partition* of S and f.

Now, suppose that S and f changes by a sequence of discrete updates. We allow three types of updates: *insertions* that add one new member x to S, with an arbitrary value of $f(x)$, *deletions* that remove an arbitrary member from S, and *changes* to the value $f(x)$ of an existing member of S. As S and f undergo a sequence of updates of these types, we wish to maintain both the h-index of S and f and a valid h-partition realizing this h-index.

To keep track of the h-index and h-partition, 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 in which the keys in each (key,value) pair are the members of H and the values are not used).
- The number h of elements in H.
- The set $B = \{x \in H \mid f(x) = |H|\}$. Intuitively, we think of these elements as being "on the bubble": if the h-index is to be reduced by one, each of these elements must be removed from H.
- A dictionary C mapping each non-negative integer i to the set $\{x \in (S \setminus B) \mid f(x) = i\}$ of the elements that f maps to i (excluding the elements already included in B). 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.

These structures allow us to insert an element x into our structure, with f-value $f(x)$, by performing the following sequence of steps:

1. Set $F[x] = f(x)$
2. If $f(x)$ is one of the existing keys of C, add x to the existing set $C[f(x)]$; otherwise, set $C[f(x)]$ to point to the new singleton set $\{x\}$.
3. Test whether $f(x) > |H|$. If not, the h-index does not change, and the insertion operation is complete.
4. If $f(x) > |H|$:

- If B is nonempty, the h-index does not change; however, x must be included in H. To do so, we choose an arbitrary $y \in B$, and remove y from B and from H. If h is one of the keys of C, we add y to the existing set $C[h]$; otherwise, we set $C[h]$ to point to the new singleton set $\{y\}$.
- Otherwise, if B is empty, the insertion causes the h-index ($|H|$) to increase by one. In this case, we increment h by setting $h := h + 1$. Additionally, we test whether the new value of h is one of the keys in C. If it is, we set B to equal the identity of the set in $C[h]$ and delete the entry for key h in C; otherwise, we set B to the empty set.

To delete x from our structure, we perform a similar sequence of operations that reverses the effect of an insertion:

1. Remove the entry for x from F.
2. If x belongs to B, remove it from B; otherwise remove it from the set $C[f(x)]$, and remove the entry for $f(x)$ in C if the removal of x causes $C[f(x)]$ to become empty.
3. If x did not belong to H, the h-index does not change, and the deletion operation is complete.
4. If x belonged to H, remove it from H, and test whether $C[h]$ is nonempty.
   - If $C[h]$ is nonempty, we move an arbitrary element y from $C[h]$ to B, and if this causes $C[h]$ to become empty, we remove the entry for h from C. Because we have replaced x with y in H, the h-index does not change.
   - If $C[h]$ is empty, the deletion causes the h-index to decrease by one, so we decrement h by setting $h := h - 1$. Additionally, if the remaining set B is nonempty, we store its identity into $C[h + 1]$. We replace B by the empty set: there may be additional elements y with $f(y) = h$, but none of them belong to the set H in the current h-partition.

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 $|H|$:

1. If x is in H and the new value of $f(x)$ is greater than or equal to h, or x is not in H and the new value of $f(x)$ is less than or equal to h, then set $F[x]$ to the new value and return.
2. Otherwise, delete x and then reinsert it with its new value.

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.*

Proof: The time analysis follows immediately from the description of the data structure update operations: each operation consists of a bounded number of updates to dictionary structures, with no looping or recursion. Additionally, these updates maintain invariant the desired properties of the set B and the dictionary of sets $C[i]$, namely that they partition S properly by their values of $f(x)$, that B consists exactly of those elements of H with $f(x) = |H|$, and that H consists of B together with those elements of S with $f(x) > |H|$.

Thus, $h = |H|$ has the property that there exists a set (namely H) with h elements, all of which have function value at least h. There can be no larger h' with the same property, because all of the elements with value greater than h belong to H already so there can be no larger set of elements with larger values. It follows that h is the correct h-index of S and f, and that $(H, S \setminus H)$ is a correct h-partition. □

As observed in the introduction, classical hash table data structures allow the dictionary operations to be implemented in constant expected time per operation. Alternatively, the dictionaries may be implemented using slower deterministic data structures such as van Emde Boaz trees.

Corollary 1 *If G is a graph, undergoing a sequence of edge insertions and deletions, we may maintain the h-index of G, and an h-partition of the vertices of G, in constant time per update.*

Proof: Each update to G is reflected in a pair of change operations to the vector of vertex degrees of G, so the result follows immediately from Theorem 1. □

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 construct a data structure that changes H so infrequently, it is necessary to restrict the set of updates that are allowed: allowing arbitrary changes to a vector of integers, as we did in the previous section, could lead to a constant rate of change to H. However, we observe that, in Corollary 1, the vector of vertex degrees does not change arbitrarily. Rather, each update causes only two vertex degrees to change, and they change only by being incremented or decremented by a single unit. To model this, we again assume a more general setup in which we are given a dynamic set S and a dynamic integer function f, but we allow f to change only by incrementing or decrementing one of its values, and we allow S to change only by inserting or deleting an element x for which $f(x) = 0$. As we now describe, a modification of the H-partition data

structure has the desired 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 $\{x \in H \setminus P \mid f(x) = i\}$.

As an accounting technique for the analysis of the algorithm (not something actually stored within our data structure) we associate a (fractional) number of "credits" with each member of P, that is zero when that element is added to P. Each increment operation adds $1/|H|^2$ credit to each current member of P, and each decrement operation on a member of P adds $1/|H|$ credits to that member.

Lemma 1 *Any sequence of operations during which $|H|$ changes from h to $h' > h$ includes at least $(h' - h)^2$ increment operations.*

Proof: There exist at least $h' - h$ members of the set H after the sequence that were not members prior to the sequence. Each of these elements has $f(x) \leq h$ prior to the sequence (else it would belong to H) and $f(x) \geq h'$ after the sequence, so the number of increments for these elements alone must have been at least $(h' - h)^2$. □

Lemma 2 *Any element x that is removed from P must have accumulated $\Omega(1)$ credits.*

Proof: Let h be the value of $|H|$ at the time x was added to P, and h' be the value of $\max(h, |H|)$ at the time it is removed. We consider two cases:

- Suppose first that, between the times x was added to P and the time it was removed, the value of $|H|$ remained always at most $2h$. Then by Lemma 1, x must have accumulated at least $(h' - h)^2/(2h)^2$ credits from increment operations. Additionally, when x was added to P, it must have been the case that $f(x) \geq 2h$, and when it was removed it must have been the case that $f(x) \leq h'$, so in between those two times it must have been the argument of at least $2h - h'$ decrement operations. Therefore, using the assumption that $|H|$ remains small throughout this sequence, it must also have received $\Omega((2h - h')/h)$ credits from decrement operations. But for any $h' \geq h$, $(h' - h)^2/h^2 + (2h - h')/h = \Omega(1)$.

- Suppose on the other hand that, at some time between the addition and the removal of x, the value of $|H|$ reached $2h$. Then, using Lemma 1 alone, x must have accumulated $(2h - h)^2/(2h)^2 = \Omega(1)$ credits within this time period.

Thus, in either case, x must have accumulated $\Omega(1)$ credits, as the lemma states. □

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.*

Proof: The number of additions to P is equal to the number of removals from P, plus the number of items that remain in P at the end of the sequence. But by Lemma 1 we can find a subsequence I of increase operations such that the final value of $|H|$ (and therefore also the number of remaining items in P) is $O(\sum_{i\in I} 1/h_i)$. Thus, we need count only the number of times elements are removed from P. By Lemma 2, this number of removals is proportional to the total number of credits that have been accumulated by all elements over the course of σ. But, since each operation assigned at most $1/h_i$ credits, this total is at most q. □

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, by Lemma 1, 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 t 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.

To insert a vertex v (with no incident edges), we perform the insertion into the gradual approximate h-partition structure of Section 3; because v has no edges, its insertion cannot change H and does not affect any of the other pieces of information described above. Similarly, to delete a vertex with no incident edges, we perform the deletion in the gradual approximate h-partition, and do not change any of our other data structures.

To insert an edge uv, we perform the following steps:

1. We look up the pair (u, v) in P, and if it is one of the keys of P we increase t by the quantity $P[u, v]$.
2. For each vertex w in H, we test whether edges uw and vw are present, and if so we increase t by one.
3. If u does not belong to H, then for each neighbor $w \neq v$ of u we increase $P[v, w]$ by one.
4. If v does not belong to H, then for each neighbor $w \neq u$ of v we increase $P[u, w]$ by one.
5. We add edge uv to E.
6. We update the degrees of u and v in the graduate approximate h-partition structure. If this update causes any change to the approximate H-partition, the change will take the form of the addition of one or more vertices to H. For each such vertex, x, we find all two-edge paths yxz connecting pairs of neighbors of x, and for each such path we decrement $P[y, z]$.

Similarly, to delete an edge uv, we perform a corresponding sequence of steps:

1. We look up the pair (u, v) in P, and if it is one of the keys of P we decrease t by the quantity $P[u, v]$.
2. For each vertex w in H, we test whether edges uw and vw are present, and if so we decrease t by one.
3. If u does not belong to H, then for each neighbor $w \neq v$ of u we increase $P[v, w]$ by one.
4. If v does not belong to H, then for each neighbor $w \neq u$ of v we increase $P[u, w]$ by one.
5. We remove edge uv from E.
6. We update the degrees of u and v in the graduate approximate h-partition structure. If this update causes any change to the approximate H-partition, the change will take the form of the removal of one or more vertices from H. For each such vertex, x, we find all two-edge paths yxz connecting pairs of neighbors of x, and for each such path we increment $P[y, z]$.

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 the time analysis for these operations follows from Corollary 1 and Theorem 2. In the remainder of the proof we concentrate 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 analyze the steps of the algorithm that 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.

Among any subset of three vertices $\{u, v, w\}$ of a graph G there are three possible edges uv, uw, and vw, and the set of these edges that are actually present in G determines 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.

If G is a dynamic graph, then it is straightforward to maintain 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. Updating these quantities takes constant time per operation: 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, $g_0 + g_1 + g_2 + g_3 = \binom{n}{3}$, follows from the fact that each triple of vertices in G forms exactly one induced subgraph of one of these four types. The equation from the second row, $g_1 + 2g_2 + 3g_3 = m(n-2)$, is a form of double counting: the left hand side of the equation counts the total number of edges in all three-vertex subgraphs in one way, by multiplying the number of edges in each type of subgraph by the number of times that subgraph appears and adding these products, while the right hand side of the equation counts the same quantity (the total number of edges in all three-vertex graphs) by multiplying the number of edges (m) by the number of three-vertex graphs each edge appears in ($n-2$). The third row's equation, $g_2 + 3g_3 = p_2$, similarly counts the total number of two-edge paths occurring in all three-vertex subgraphs: the left hand side again multiplies the number of paths per subgraph type by the number of subgraphs with that type, and the right hand side is just the number of two-edge paths, as each path appears in exactly one three-vertex subgraph. The fourth equation $g_3 = c_3$ follows from the fact that 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 next section, 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 Subgraphs with more than three vertices

If $s_i = s_i(G)$ denote the number of star subgraphs $K_{1,i}$ in G, we may maintain s_i, for any constant i, in constant time per update, as it is a sum of polynomials of the vertex degrees: $s_i = \sum_v d_v(d_v-1)\cdots(d_v-i-1)/i!$. For instance, the number of claws (three-leaf stars) in G is $s_3 = \sum_v d_v(d_v-1)(d_v-2)/6$. In at least one other nontrivial case we may maintain the number of four-vertex subgraphs of a certain type as efficiently as the number of triangles.

Theorem 4 *We may maintain a dynamic graph subject to edge insertions and deletions and to insertions and deletions of isolated vertices, and keep track of the number p_3 of four-vertex paths in the graph, in amortized time $O(h)$ per update where h is the h-index of the graph at the time of an update.*

Proof: Let q denote the number of sequences of three edges that form either a path or a cycle in G. Let d_v denote the degree of v (that is, its number of incident edges), and let P_v denote the number of two-edge paths having v as an endpoint (that is, $\sum(d_w - 1)$ where the sum is over all neighbors of v in G).

Inserting an edge uv into the graph G increases q by $d_u d_v + P_u + P_v$: the term $d_u d_v$ counts the paths with uv as middle edge, and the other two terms count the paths having v or u as endpoint. Similarly, removing edge uv decreases q by $(d_u-1)(d_v-1) + (P_u - d_v + 1) + (P_v - d_u + 1)$. Thus, if we can calculate P_u and P_v, we can correctly update q.

Our data structure stores the numbers d_v for each vertex v, and the numbers P_u only for those vertices u that belong to the set H maintained by the gradual partition of Section 3. When a vertex is added to H, the value P_u stored for it may be computed in time $O(h)$. When we insert or delete an edge uv, the

numbers P_u and P_v that we need to use to update q may be found either by looking them up in this data structure (if the endpoints u or v of the updated edge belong to H) or in time $O(h)$ by looking at all neighbors of the endpoints if they do not belong to H. Finally, whenever we insert or delete an edge uv, we must update the numbers P_w for all vertices w belonging to H, where either w is one of the two endpoints u and v or it is adjacent to one or both of these endpoints; this update may be performed in constant time per member of H, or $O(h)$ time total.

The number of four-vertex paths that we maintain is then $p_3 = q - 3c_3$ where c_3 denotes the number of triangles in the graph as maintained by our other structures. □

The counts of larger subgraphs in G obey additional linear relations: for instance, $\sum_v P_v^2 = p_4 + 2p_2 + 3s_3 + 4c_4$. Based on this principle, in work subsequent to the original presentation of the results in this paper, we extended our algorithms to count all possible four-vertex subgraphs [17]. However, the time per update for these four-vertex subgraph counting algorithms is $O(h^2)$, higher than the time for the data structures in this paper.

7 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 5 *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 [27], 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 6 *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.

8 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, for a given number of vertices and edges, 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, 27, 30, 31]. Typically, γ lies in

n	h	$\log n$	$\log h$	$\frac{\log h}{\log n}$
10	5	2.3026	1.6094	0.6990
10	10	2.3026	2.3026	1.0000
11	6	2.3979	1.7918	0.7472
11	6	2.3979	1.7918	0.7472
12	2	2.4849	0.6931	0.2789
13	2	2.5649	0.6931	0.2702
16	6	2.7726	1.7918	0.6462
16	6	2.7726	1.7918	0.6462
16	8	2.7726	2.0794	0.7500
16	7	2.7726	1.9459	0.7018
17	8	2.8332	2.0794	0.7340
18	4	2.8904	1.3863	0.4796
19	7	2.9444	1.9459	0.6609
21	14	3.0445	2.6391	0.8668
21	9	3.0445	2.1972	0.7217
21	4	3.0445	1.3863	0.4553
23	8	3.1355	2.0794	0.6632
24	10	3.1781	2.3026	0.7245
24	8	3.1781	2.0794	0.6543
24	7	3.1781	1.9459	0.6123
24	7	3.1781	1.9459	0.6123
25	16	3.2189	2.7726	0.8614
26	5	3.2581	1.6094	0.4940
27	12	3.2958	2.4849	0.7540
31	7	3.4340	1.9459	0.5667
32	9	3.4657	2.1972	0.6340
32	28	3.4657	3.3322	0.9615
32	30	3.4657	3.4012	0.9814
32	18	3.4657	2.8904	0.8340
33	10	3.4965	2.3026	0.6585
34	34	3.5264	3.5264	1.0000
34	34	3.5264	3.5264	1.0000
35	10	3.5553	2.3026	0.6476
35	12	3.5553	2.4849	0.6989

n	h	$\log n$	$\log h$	$\frac{\log h}{\log n}$
35	12	3.5553	2.4849	0.6989
35	7	3.5553	1.9459	0.5473
35	14	3.5553	2.6391	0.7423
35	12	3.5553	2.4849	0.6989
36	4	3.5835	1.3863	0.3869
36	9	3.5835	2.1972	0.6131
36	8	3.5835	2.0794	0.5803
37	11	3.6109	2.3979	0.6641
37	11	3.6109	2.3979	0.6641
37	12	3.6109	2.4849	0.6882
38	4	3.6376	1.3863	0.3811
39	10	3.6636	2.3026	0.6285
39	10	3.6636	2.3026	0.6285
39	12	3.6636	2.4849	0.6783
39	18	3.6636	2.8904	0.7890
39	20	3.6636	2.9957	0.8177
39	12	3.6636	2.4849	0.6783
41	10	3.7136	2.3026	0.6200
44	16	3.7842	2.7726	0.7327
44	23	3.7842	3.1355	0.8286
46	12	3.8286	2.4849	0.6490
46	17	3.8286	2.8332	0.7400
48	33	3.8712	3.4965	0.9032
48	33	3.8712	3.4965	0.9032
48	17	3.8712	2.8332	0.7319
54	15	3.9890	2.7081	0.6789
58	47	4.0604	3.8501	0.9482
58	58	4.0604	4.0604	1.0000
59	28	4.0775	3.3322	0.8172
60	8	4.0943	2.0794	0.5079
60	8	4.0943	2.0794	0.5079
62	14	4.1271	2.6391	0.6394
64	8	4.1589	2.0794	0.5000
65	10	4.1744	2.3026	0.5516

Table 1: Raw data from analysis of real-world networks, part I

n	h	$\log n$	$\log h$	$\frac{\log h}{\log n}$
69	27	4.2341	3.2958	0.7784
69	27	4.2341	3.2958	0.7784
69	27	4.2341	3.2958	0.7784
71	22	4.2627	3.0910	0.7251
71	22	4.2627	3.0910	0.7251
72	7	4.2767	1.9459	0.4550
73	6	4.2905	1.7918	0.4176
75	8	4.3175	2.0794	0.4816
75	8	4.3175	2.0794	0.4816
80	7	4.3820	1.9459	0.4441
80	24	4.3820	3.1781	0.7252
84	8	4.4308	2.0794	0.4693
86	10	4.4543	2.3026	0.5169
97	35	4.5747	3.5553	0.7772
97	35	4.5747	3.5553	0.7772
100	11	4.6052	2.3979	0.5207
100	20	4.6052	2.9957	0.6505
101	14	4.6151	2.6391	0.5718
101	41	4.6151	3.7136	0.8047
102	13	4.6250	2.5649	0.5546
105	5	4.6540	1.6094	0.3458
111	8	4.7095	2.0794	0.4415
112	6	4.7185	1.7918	0.3797
118	6	4.7707	1.7918	0.3756
124	116	4.8203	4.7536	0.9862
124	6	4.8203	1.7918	0.3717
128	38	4.8520	3.6376	0.7497
128	38	4.8520	3.6376	0.7497
128	38	4.8520	3.6376	0.7497
129	18	4.8598	2.8904	0.5947
151	37	5.0173	3.6109	0.7197
154	6	5.0370	1.7918	0.3557
169	7	5.1299	1.9459	0.3793
180	7	5.1930	1.9459	0.3747

n	h	$\log n$	$\log h$	$\frac{\log h}{\log n}$
205	11	5.3230	2.3979	0.4505
234	3	5.4553	1.0986	0.2014
244	11	5.4972	2.3979	0.4362
265	8	5.5797	2.0794	0.3727
275	6	5.6168	1.7918	0.3190
311	13	5.7398	2.5649	0.4469
332	48	5.8051	3.8712	0.6669
332	12	5.8051	2.4849	0.4281
352	7	5.8636	1.9459	0.3319
395	19	5.9789	2.9444	0.4925
452	10	6.1137	2.3026	0.3766
489	16	6.1924	2.7726	0.4477
533	12	6.2785	2.4849	0.3958
638	15	6.4583	2.7081	0.4193
673	13	6.5117	2.5649	0.3939
674	10	6.5132	2.3026	0.3535
719	13	6.5779	2.5649	0.3899
775	14	6.6529	2.6391	0.3967
1022	27	6.9295	3.2958	0.4756
1059	37	6.9651	3.6109	0.5184
1096	13	6.9994	2.5649	0.3665
1490	96	7.3065	4.5643	0.6247
1577	22	7.3633	3.0910	0.4198
1882	14	7.5401	2.6391	0.3500
2361	56	7.7668	4.0254	0.5183
2361	56	7.7668	4.0254	0.5183
2361	56	7.7668	4.0254	0.5183
2909	60	7.9756	4.0943	0.5134
3084	38	8.0340	3.6376	0.4528
4470	47	8.4051	3.8501	0.4581
6927	88	8.8432	4.4773	0.5063
7343	65	8.9015	4.1744	0.4690
8497	34	9.0475	3.5264	0.3898
10616	25	9.2701	3.2189	0.3472

Table 2: Raw data from analysis of real-world networks, part II

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.

8.1 Corpus of real-world graphs

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 [22], statistical modeling software in R. statnet includes ERGM functionality, making it a good example for data used specifically in the context of ERGM models. Finally, we included data available on the UCI Network Data Repository [9], including some larger networks such as the WWW, weblog 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 Tables 1 and 2.

8.2 Summary statistics

A summary of the statistics for network size and h-index for this sample of 136 real-world networks are in Table 3, below. 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 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.

	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

Table 3: Summary statistics for real-world network data

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 next section.

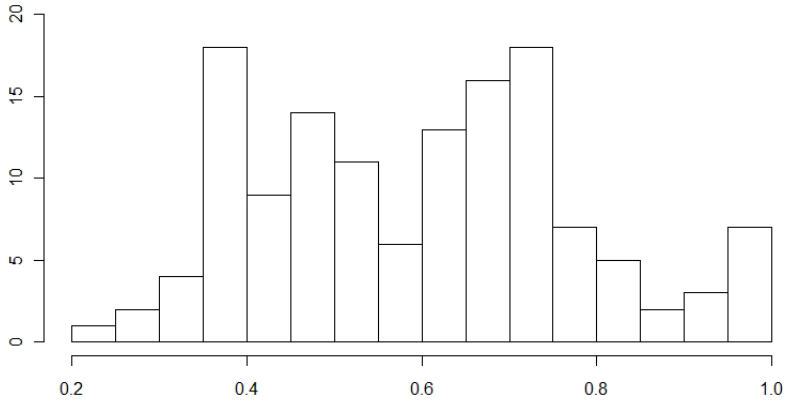

Figure 1: A frequency histogram for $\log h / \log n$.

8.3 Detailed analysis of real-world network data

We calculated the h-index of the networks in our sample in **R**, using a subroutine provided by Carter T. Butts, one of the authors of the statnet software suite. The data that results from this calculation in plotted in Figure 2. This figure suggests that the data might be more appropriately viewed on a log-log scale. This plot is seen in Figure 3.

To find an upper bound on the scaling of the h-index of our real-world networks we clustered the data into two groups, and used quantile regression to

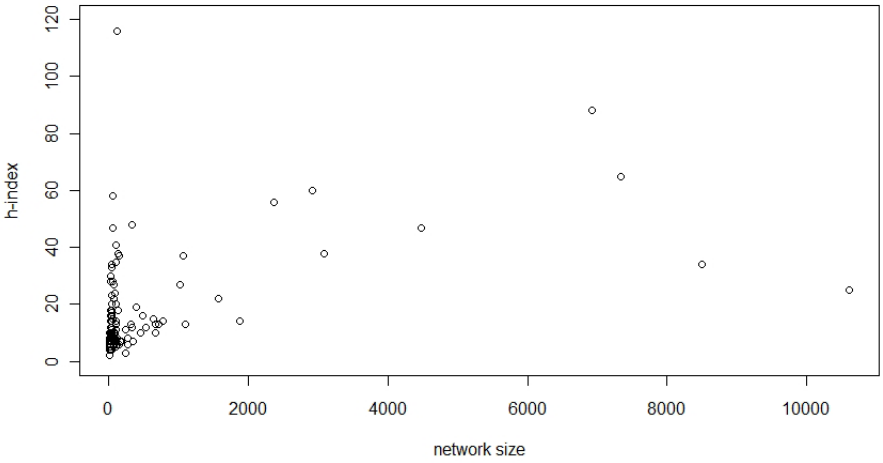

Figure 2: Scatter plot of h-index and network size

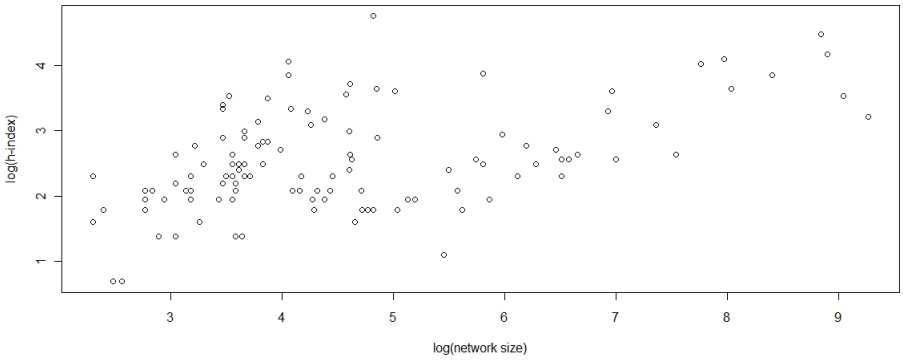

Figure 3: Scatter plot of h-index and network size, on log-log scale

fit the data with curves of the form $\log h = \beta_0 + \beta_1 \log n$, at the 95th percentile. That is, we are looking for a power law $h = cn^{\beta_1}$, and we want 95% of the graphs to have an h-index no larger than the one predicted by this law. We fit a law of this type to the two clusters separately to provide a more conservative and substantive prediction. The resulting regression lines are reported in Table 4. Corresponding goodness of fit measure are also reported in Table 5. We note that these are conservative estimates and the actual scaling is likely better.

Cluster	Intercept β_0	Slope β_1	df
1	0.0609	0.9735	92
	(-0.964, 2.581)	(0.231, 1.266)	
2	-0.598	0.604	44
	(-1.938, 5.248)	(0.44712, 0.847)	

Table 4: Coefficients for quantile regression lines

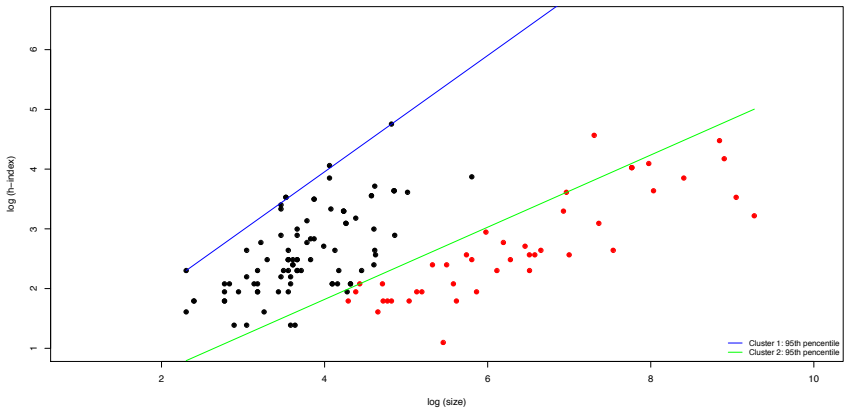

Figure 4: h-index scaling using quantile regression fits

Cluster	log-like	AIC	BIC
1	-109.345	222.691	227.734
2	-41.071	86.143	89.712

Table 5: Goodness of fit measures for quantile regression lines

9 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* and the *degeneracy*, two graph invariants that are within a constant factor of each other and that have both used in static subgraph isomorphism and clique-finding algorithms [7, 13]. Can we speed up our dynamic algorithms to run more quickly on graphs in which these parameters are bounded?

The algorithms in this paper can handle only undirected graphs, but the directed case is also of interest: for instance, in social networks, not all ties between actors are bidirectional. In subsequent work, we extended the algorithms here to count three-vertex directed subgraphs in the same $O(h)$ time bound per update [17], as well as to count larger subgraphs such as 4-cycles, 4-cliques, and claws in a time bound that is be slower than our triangle-finding algorithms but that may still provide speedups over static algorithms. Lin, Soulignac, and Szwarcfiter provide a simpler data structure that handles vertex insertion and deletion updates in time $O(dh)$ per update, where d is the degree of the updated vertex; however, their method is less efficient for edge updates [28].

The h-index of a graph, and the partition of a graph into high and low degree vertices based on the h-index, may be of interest in other graph algorithms as well. For instance, subsequently to our work the same decomposition has been used by Cheng et al. in an external-memory algorithm for listing maximal cliques in real-world graphs [6].

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 research in implementing our data structures and testing their efficiency in practice.

References

[1] R. Adler, J. Ewing, and P. Taylor. *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.* Joint Committee on Quantitative Assessment of Research, 2008.

[2] R. Albert, H. Jeong, and A.-L. Barabási. Diameter of the world wide web. *Nature* 401:130–131, 1999, doi:10.1038/43601, arXiv:cond-mat/9907038.

[3] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica* 17(3):209–223, 1997, doi:10.1007/BF02523189.

[4] V. Batagelj and A. Mrvar. Pajek datasets. Web page http://vlado.fmf.uni-lj.si/pub/networks/data/, 2006.

[5] S. P. Borgatti, M. G. Everett, and L. C. Freeman. *UCINet 6 for Windows: Software for social network analysis.* Analytic Technologies, Harvard, MA, 2002.

[6] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.* 36(4):21:1–21:34, 2011, doi:10.1145/2043652.2043654.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14(1):210–223, 1985, doi:10.1137/0214017.

[8] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.* 9(3):251–280, 1990, doi:10.1016/S0747-7171(08)80013-2.

[9] C. L. DuBois and P. Smyth. UCI Network Data Repository. Web page http://networkdata.ics.uci.edu, 2008.

[10] R. A. Duke, H. Lefmann, and V. Rödl. A fast approximation algorithm for computing the frequencies of subgraphs in a given graph. *SIAM J. Comput.* 24(3):598–620, 1995, doi:10.1137/S0097539793247634.

[11] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.* 326(1–3):57–67, 2004, doi:10.1016/j.tcs.2004.05.009.

[12] D. Eppstein. Connectivity, graph minors, and subgraph multiplicity. *J. Graph Theory* 17:409–416, 1993, doi:10.1002/jgt.3190170314.

[13] D. Eppstein. Arboricity and bipartite subgraph listing algorithms. *Inform. Process. Lett.* 51(4):207–211, August 1994, doi:10.1016/0020-0190(94)90121-X.

[14] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms Appl.* 3(3):1–27, 1999, arXiv:cs.DS/9911003.

[15] D. Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica* 27:275–291, 2000, doi:10.1007/s004530010020, arXiv:math.CO/9907126.

[16] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[17] D. Eppstein, M. T. Goodrich, D. Strash, and L. Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theor. Comput. Sci.* 447:44–52, 2012, doi:10.1016/j.tcs.2011.11.034, arXiv:1009.0783.

[18] J. Feigenbaum and S. Kannan. Dynamic graph algorithms. *Handbook of Discrete and Combinatorial Mathematics*. CRC Press, 2000.

[19] O. Frank. Statistical analysis of change in networks. *Stat. Neerl.* 45:283–293, 199, doi:10.1111/j.1467-9574.1991.tb01310.x.

[20] O. Frank and D. Strauss. Markov graphs. *J. Amer. Statistical Assoc.* 81:832–842, 1986, `http://www.jstor.org/stable/2289017`.

[21] E. Gilbert. Random Graphs. *Ann. Math. Stat.* 30:1141–1144, 1959, doi:10.1214/aoms/1177706098.

[22] M. S. Handcock, D. Hunter, C. T. Butts, S. M. Goodreau, and M. Morris. statnet: An R package for the Statistical Modeling of Social Networks. Web page http://www.csde.washington.edu/statnet, 2003.

[23] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proc. Natl. Acad. Sci. USA* 102(46):16569–16572, 2005, doi:10.1073/pnas.0507655102.

[24] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.* 7(4):413–423, 1978, doi:10.1137/0207033.

[25] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics* 20(11):1746–1758, 2004, doi:10.1093/bioinformatics/bth163.

[26] T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. *Inform. Process. Lett.* 74(3–4):115–121, 2000, doi:10.1016/S0020-0190(00)00047-8.

[27] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature* 411:907–908, 2001, doi:10.1038/35082140, arXiv:cond-mat/0106507.

[28] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter. Arboricity, h-index, and dynamic algorithms. *Theoret. Comput. Sci.* 426–427:75–90, 2012, doi:10.1016/j.tcs.2011.12.006, arXiv:1005.2211.

[29] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Comment. Math. Univ. Carol.* 26(2):415–419, 1985.

[30] M. E. J. Newman. The structure and function of complex networks. *SIAM Review* 45:167–256, 2003, doi:10.1137/S003614450342480, arXiv:cond-mat/0303516.

[31] D. J. d. S. Price. Networks of scientific papers. *Science* 149(3683):510–515, 1965, doi:10.1126/science.149.3683.510.

[32] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22(8):974–980, 2006, doi:10.1093/bioinformatics/btl030.

[33] G. Robins and M. Morris. Advances in exponential random graph (p^*) models. *Social Networks* 29(2):169–172, 2007, doi:10.1016/j.socnet.2006.08.004. Special issue of journal with four additional articles.

[34] T. A. B. Snijders. Markov chain Monte Carlo estimation of exponential random graph models. *Journal of Social Structure* 3(2):1–40, 2002, `http://www.cmu.edu/joss/content/articles/volume3/Snijders.pdf`.

[35] T. A. B. Snijders, P. E. Pattison, G. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological Methodology* 36(1):99–153, 2006, doi:10.1111/j.1467-9531.2006.00176.x.

[36] M. Thorup and D. R. Karger. Dynamic graph algorithms with applications. *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT 2000)*, pp. 667–673. Springer-Verlag, Lecture Notes in Computer Science 1851, 2000, doi:10.1007/3-540-44985-X_1.

[37] V. Vassilevska and R. Williams. Finding, minimizing and counting weighted subgraphs. *Proc. 41st ACM Symposium on Theory of Computing*, 2009, doi:10.1145/1536414.1536477.

[38] S. Wasserman and P. E. Pattison. Logit models and logistic regression for social networks, I: an introduction to Markov graphs and p^*. *Psychometrika* 61:401–425, 1996, doi:10.1007/BF02294547.

[39] R. Yuster. Finding and counting cliques and independent sets in r-uniform hypergraphs. *Inform. Process. Lett.* 99(4):130–134, 2006, doi:10.1016/j.ipl.2006.04.005.