Fast Approximation of the "Neighbourhood" Function for Massive Graphs

Christopher R. Palmer
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
crpalmer@cs.cmu.edu

Phillip B. Gibbons
Information Sciences Research Center
Bell Laboratories
Murray Hill, NJ
gibbons@research.bell-labs.com

Christos Faloutsos
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA
christos@cs.cmu.edu

Abstract

The neighbourhood function, $N(h)$, is the number of pairs of nodes that are within distance h. The neighbourhood function provides useful information for graphs such as the structure of XML documents, OODBs, Web graphs, telephone (caller-callee) records, citation graphs, to name a few. It is *very* expensive to compute the neighbourhood function exactly on large graphs. Instead, we propose an algorithm using approximate counting. Our approximation runs with any amount of available memory, providing excellent scalability to arbitrarily large graphs. By applying our approximation algorithm on large real and synthetic graphs, we find that our approximation algorithm is significantly faster and more accurate than the best prior approximation scheme. Our approximation runs over 400 times faster than the exact computation (with less than a 10% error).

1 Introduction

"What is the diameter of the Web?"

"Are the telephone caller-callee graphs in the U.S. similar to the ones in Europe?"

"Is the citation graph for Physics different from the one for Computer Science?"

"Are users in India further away from the bulk of the Web than those in the U.S.?"

These are some of the questions that the "neighbourhood" function can help us answer. The *neighbourhood function*, $N(h)$, of a graph $G = (V, E)$ is the number of pairs of nodes that are within

distance h. More formally, for all nodes u and v in V, let $\text{dist}(u, v)$ be the number of edges in the shortest path in G from u to v. We define:

$$S(u, h) = \{(u, v) \in V \times V : \text{dist}(u, v) \leq h\}$$

$$S(h) = \bigcup_{u \in V} S(u, h)$$

$$N(h) = |S(h)|$$

where $S(u, h)$ is the set of neighbours of u that are at most distance h away, $S(h)$ is the neighbourhood set for distance h (the pairs of neighbours that are at most distance h apart) and $N(h)$ is the neighbourhood function. The graph can be directed or undirected.

The neighbourhood function is very important in databases. We find that there are a wealth of current and new applications of the neighbourhood function:

- *Path query optimization.* Both OODMS and XML query optimizers need to evaluate the cost of *path-queries* [12, 8]. The cost of expanding a path query up to h steps is analogous to the neighbourhood function. Our approximation can be used to compute accurate path statistics which may be able to significantly improve these optimizers.

- *Understanding our data.* Researchers are currently using the neighbourhood function as a tool to understand the Internet and the Web [6, 2, 10]. It is critical that we understand the properties of these graphs if we want to build the best databases to manage their content, for understanding caching properties of the access patterns, etc. Recent work attempting to understand the structure of the Web uses $|S(u, \infty)|$ [3, 10]. Finally, taking a social networks perspective, the distribution of minimum path lengths (which we can compute from the neighbourhood function) is used to identify the "small-world" phenomenon in the Web [1].

- *Modelling our data.* Similarly, the neighbourhood function is necessary to model accesses to graph data, such as the set of all phone calls or citations within Computer Science papers [5]. In this case, a reliable neighbourhood function is necessary to justify analytic performance analysis.

- *Transitive closures.* Since $N(\infty)$ is the transitive closure, we are also proposing a new approximation algorithm for estimating the size of a datalog query [11].

- *Graph features (indexing, data-mining).* In the discussion section of this paper, we will consider using graph metrics to define features for a graph. These features allow us to store

graphs into a spatial index and pose nearest neighbour queries. While these results are preliminary, they are indicative of the new applications that can be considered now that it is possible to efficiently compute the neighbourhood function for large graphs.

To meet all of these applications, we can identify certain key properties that a good neighbourhood function (or approximation) should satisfy:

- *Error guarantees:* must estimates $N(h)$ for all h with high accuracy.
- *Fast:* must scale linearly with number of nodes and edges.
- *Low storage requirements:* For graphs with n nodes, the additional memory used should be $O(n)$.
- *Adapts to the available memory:* $O(n)$ memory may be too much for the Web (with billions of nodes). Must be able to use $O(n)$ external storage and any (much smaller) amount of RAM.
- *Parallelizable:* To handle large graphs efficiently, distributing the work over multiple processors and/or multiple workstations in a network is essential.
- *Sequential scans of the edges:* Accessing the edges in random order results in terrible performance for the common case that the set of edges does not fit in memory. Using sequential scans solves this problem as the edges can be stored in any number of ways in secondary storage.
- *Estimates $|S(u,h)|$:* In addition to estimating $N(h)$, certain applications require accurate estimates for $|S(u,h)|$ for some, or all, of the nodes u.

The main contribution of this paper is to develop an algorithm in section 3 that meets all of these requirements. However, before proposing the algorithm we provide a brief survey of the related work in section 2. After proposing our algorithm and arguing that it meets all of the requirements, we will experimentally evaluate it in section 4. By using a combination of 3 real data sets and 4 synthetic data sets, we will verify our analytic discussion and the experimental evaluation also allows us to compare our algorithm with the other known neighbourhood function approximation algorithm. Using these data sets, we find that the expected behaviour is much better than any other approximation scheme. Also in our experimental evaluation, we will consider different methods of sampling to determine that they are all inferior to our approximation scheme (because they lack the accuracy guarantees and/or because they require the edge file to be in memory and consequently do

not scale with the number of edges). Finally, experimentally, we will find that our approximation can be over 400x faster than the exact computation and that it is both much faster and more accurate than any other known approximation scheme.

After our thorough experimental evaluation, we present a discussion section in which we introduce some ways of using the neighbourhood function. In particular, we look at ways in which the individual neighbourhood functions can be used to glean information about the nodes in a graph. Also, we look at ways in which the neighbourhood function can be used to define attributes for graphs, allowing for similarity computations. Finally, in section 6 we conclude.

2 Related Work

Computing the neighborhood function $N(h)$ is trivial for $h = 0$ and $h = 1$: It is $|V|$, and either $|V| + |E|$ for directed graphs or $|V| + 2|E|$ for undirected graphs, respectively. For $h = 2$, $N(h)$ is reminiscent of the size of the (self-)join of the edge relation. Specifically, if each edge is viewed as a tuple with attributes "first" and "second", then $N(2) - N(1)$ is the result size for the query:

```
select distinct E1.first, E2.second
from edge-rel E1, edge-rel E2
where E1.second = E2.first
```

The key to computing $N(h)$ for any $h \geq 2$ is the "distinct" here: one must not count the duplicates (i.e., the multiple paths between two nodes).

There are a number of different approaches for computing the neighbourhood function for $h \geq 2$. An obvious approach is to repeatedly multiply the graph's adjacency matrix. With a fast matrix multiply package and the standard repeated squaring trick, this takes $O(n^{2.81})$ time for an n-node graph (asymptotically, it can be done in $O(n^{2.38})$ time). Unfortunately, it also takes $O(n^2)$ memory and this is prohibitive. To reduce the amount of memory that is needed, it is typical to use a breadth first search of the graph. A breadth-first search beginning from u can be used to easily compute $S(u, h)$ for all h. We can compute $N(h)$ for all h by running a breadth-first search from each node u and then:

$$N(h) = \sum_{u \in V} |S(u, h)|$$

This takes only $O(n+m)$ storage for a graph with n nodes and m edges. The worst case running time of this algorithm is $O(nm)$. Moreover, for large graphs that are stored on disk, breadth first search results in an expensive random-like access of disk blocks. This appears to be the state of the

art solution for exact computation of $N(h)$, and we will use this as our reference implementation. Each of these algorithms produces $N(h)$ for all h.

The case of $N(\infty)$ is the size of the transitive closure. If d is the diameter of the graph, then $N(\infty) = N(d)$. Lipton and Naughton [11] presented an $O(n\sqrt{m})$ time algorithm for approximating the size of the transitive closure. They present an adaptive sampling algorithm which randomly selects starting nodes for a breadth-first search. We will look at the problems involved in using a sampling based approach in the experimental evaluation of our system. Briefly, sampling scales very poorly to large graphs because the access to the edge file is in a random order and, when the edge file no longer fits in memory, the paging behaviour is atrocious. Moreover, the quality of a sampling based approximation can be quite poor. We will show an example of a graph such that a 15% sample does not provide a useful approximation.

Lipton and Naughton's work was improved by Cohen, who gave an $O(m)$ time algorithm, which uses only $O(n + m)$ memory [4]. Cohen also presented an $O(m \log n)$ time algorithm for estimating the individual neighbourhood functions $|S(u, h)|$ for all nodes u and all h. To the best of our knowledge, this is the only algorithm which provides an approximation to the neighbourhood function. More details of this algorithm are provided when we experimentally compare it to our approximation (section 4.1.2). We will use this as a reference for assessing the speed and quality of our approximation scheme. However, Cohen's algorithms are ill-suited for large graphs due to their extensive use of random-like access (for breadth first search, heap data structures, etc.).

3 Proposed Method

We have introduced several variables throughout the proceeding discussion and will introduce several others while developing our approximation algorithm. For convenience, all symbols are defined in Table 1. This table also indicates the two parameters that are required for our approximation.

In this section, we are going to develop two algorithms based on the following approach. We first observe that $S(u, i)$ can be computed by taking each edge (u, v) and appending the paths of length $i - 1$ or less starting from v. That is, define

$$S(u, 0) = \{u : u \in V\}$$

and then by iterating over the edge set for the i^{th} time, we can compute $S(u, i)$:

$$S(u, i) = \{v' : (u, v) \in E \text{ and } v' \in S(v, i - 1)\}$$

Thus, by processing the edge file h times, we can compute the values $S(0), S(1), \ldots, S(h)$. This approach is completely impractical because the memory and time requirements to maintain the

Symbol	Parameter?	Description
n		Number of nodes
m		Number of edges
d		Diameter of the graph
r	Yes	"Extra" bits in bitmask
k	Yes	Number of parallel trials
(p_1, p_2)		Number of edge file partitions
$S(u, h)$		Nodes reachable from u in at most h steps
$M(u, h)$		Bitmask approximate representation of $\|S(u, h)\|$
$S(h)$		Pairs of nodes within h steps
$N(h)$		Number of such pairs
OR		Bitwise "or" operation
$bias$		Bias factor in the approximation $(1 + .31/k)$

Table 1: Symbols used in this paper

sets are too large. Instead, the algorithm we propose uses approximate counting to replace the set operations in this approach. This generates an approximation of the neighbourhood function.

3.1 Approximate Counting

Approximate counting algorithms are used to approximate the number of distinct items in a multi-set. Two different methods that have been proposed in the literature. The first, which we call the BitMask (BM) approach was proposed by Flajolet and Martin [7]. The second, which we call the Random Interval (RI) approach was proposed by Cohen [4].

Both algorithms solve the same problem: given a set X of N elements and a multi-set M of elements drawn from X, estimate the number of distinct elements in M.

The RI method works as follows. Each of the n elements in X is assigned a uniform random number in $[0, 1]$. Each element in M is processed sequentially and the least random number assigned to an element is recorded, say ri. Then the estimate is $\frac{1}{ri} - 1$. For example, if $X = \{1, 2, 3, 4\}$ we might assign the random numbers $r_1 = \frac{1}{2}$, $r_2 = \frac{2}{3}$, $r_3 = \frac{4}{5}$ and $r_4 = \frac{1}{3}$. When we process the multi-set $\{2, 2, 4, 2, 4, 4, 4, 2\}$ the least r_i will be recorded. In this case that is $r_4 = \frac{1}{3}$. And the estimate is $3 - 1 = 2$.

The BM approach works as follows. Given a user parameter, r, we will approximate the number of elements in M using $\log n + r$ bits (where r is a user specified parameter, usually around 5-7). For each element in X, we randomly assign one of the $\log n + r$ bits using an exponential distribution (about half the nodes will be assigned to bit 0, a quarter to bit 1, etc.). Begin with an empty bitmask and then for each element $x \in M$, we look-up the bit assigned to x and set that bit in our

bitmask. After processing all elements of M, we find the lowest order zero bit, b, in the bitmask and then our estimate is $O(2^b)$. To reduce the variance in the estimate, we can repeat this procedure for k random trials and compute the average

$$\bar{b} = \frac{b_1 + b_2 + \cdots + b_k}{k}$$

and then the estimate is $O(2^b)$. The exact value is:

$$\frac{2^b}{.7731 \cdot bias}$$

where $bias$ can be approximated as $1 + .31/k$. That is, for large k the estimate is unbiased and correction factor must be applied for small k. There are provable bounds on the quality of this estimation.

Finally, we are going to need the following observation. If we have multi-sets M_1 and M_2 and run the BM algorithm that computes bitmasks bm_1 and bm_2 then (bm_1 OR bm_2) is equal to the bitmask generated for the multi-set $M_1 \cup M_2$. That is, "bitwise-or" can be used as a union operator over the bitmask representation of the multi-sets.

3.2 Our ANF Algorithms

In this section, we develop two Approximate Neighbourhood Function (ANF) algorithms. The first works entirely in-core. We will then extend this algorithm to handle out-of-core processing (our second algorithm). It is this second version that we will use in our experimental evaluation, but we develop both as it eases the presentation.

In-core ANF Algorithm We begin by re-examining the basic neighbourhood computation:

$$S(u, 0) = \{u : u \in V\}$$

$$S(u, i) = \{v' : (u, v) \in E \text{ and } v' \in S(v, i-1)\}$$

To estimate the neighbourhood function quickly with limited memory, we are going to replace the expensive set operations with fast approximate counting operations. That is, for each node, u, and distance, h, we let $M(u, h)$ be the concatenation of k bitmasks each of which contains $\log n + r$ bits. This is a set of k BM bitmasks (which we may also treat as a single long bitmask when performing the iterations). Then we can compute the neighbourhood function using the algorithm in Figure 1.

```
FOR each node, u DO
   M(u,0) = concatenation of k bitmasks, each with 1 bit set
            (according to an exponential distribution)
DONE

FOR each distance, it, starting with 1 DO
    FOR each node, u DO M(u,it) = M(u,it-1)
    FOR each edge (u,v) DO M(u,it) = (M(u,it) OR M(v,it-1))

    The estimate is: SUM(all u) (2^b)/(.7731*bias)                        (*)
        where b is the average of the least zero bits in the k bitmasks   (*)
DONE
```

Figure 1: In-Core Approximate Neighbourhood Function (ANF)

In the course of running this algorithm, we only actually needed the current and the previous iteration. That is, for each node, u, we only need $M(u, it)$ and $M(u, it-1)$. To produce the neighbourhood function thus only requires $O(n)$ additional memory. Due to the fact that OR is equivalent to the union operation on sets, it should be obvious that $M(u, h)$ is a bitmask that can be used to estimate $|S(u, h)|$ using the BM procedure. Then, since

$$N(h) = \sum_{u \in E} |S(u, h)|$$

we see that this algorithm is producing an approximation of the neighbourhood function. The two inner loops are $O(n)$ and $O(m)$ and the outer loop runs $O(d)$ times, making the running time $O((n+m)d)$.

Our in-core approximate neighbourhood function meets most of the requirements for a good neighbourhood function:

- *Has error guarantees:* there are provable bounds on the errors.
- *Is fast:* the running time is $O((n+m)d)$ which should be quite fast since d is typically small. We will experimentally verify this intuition.
- *Has low storage requirements:* $O(n)$ additional memory.
- *Adapts to the available memory?* No! So far, our algorithm will exhibit terrible behaviour when there is less memory available than we need to represent $M(u, it)$ and $M(u, it-1)$.
- *Is easily parallelizable:* Partition the nodes among the processors. Each processor may independently compute $M(u, it)$ for each u in its set. Synchronization is only needed after each iteration.

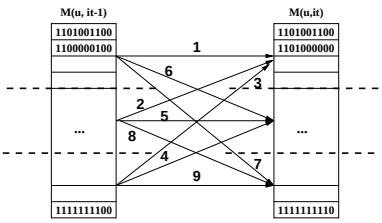


Figure 2: Partitioning the edges 3 ways gives 9 edge sets

- *Sequential scans of the edge file:* Yes.
- *Estimates* $|S(u,h)|$*:* Yes, with provable accuracy.

External version of ANF Algorithm We see that our first algorithm has achieved all our goals except that it does not have good out-of-core behaviour. To solve this problem, we need to look at the pattern of data accesses. Each edge is processed in turn. For an edge (u,v), we read the value in $M(u,it-1)$ and read and write the value in $M(v,it)$. One possible solution is to sort the edges $\{(u_i,v_i)\}$ first by v_i and then to break ties by u_i. With this sorting, we need only one entry of the $M(u,it)$ and $M(u,it-1)$ tables in memory at once. It would be easy to stream through the edges and the two tables. Unfortunately, it takes $\Theta(n \log n)$ time to sort these edges which is prohibitive. Instead, we will adopt the same idea but split the table $M(u,it-1)$ into k_1 buckets and the table $M(u,it)$ into k_2 buckets. For small values of k_1 and k_2, we can partition the edges into $k_1 k_2$ buckets simply by iterating over them once[1]. For an example of the buckets, see Figure 2 where we have partitioned each of the tables into 3 pieces which defines the 9 sets of edges (the labelled arrows in the picture).

Once we have broken the two tables into a set of buckets, we can use our idea of streaming through the data and the tables. We would begin by loading the first bucket of each table into memory and processing the edges that have one end-point in each of these buckets. The numbers on the edges in this picture show the order in which we process the edges that connect the pair of buckets. We want to use asynchronous I/O to hide as much of the paging cost as possible. We need to double buffer each of these tables. That is, first the $M(u,it-1)$ table, we need to have a buffer to hold the bucket that we are currently processing and one into which we can prefetch the next bucket to be processed. For the $M(u,it)$ table, we still only need 2 buffers. For example, when we begin processing the second bucket of this table, we need one buffer to hold the data that we are currently processing and one to hold the data that we just finished modifying (the first bucket).

[1]As long as k_1 and k_2 are less than the maximum number of files that may be opened in the system, we can partition the edges using only 2 scans of the edge file.

After this data has been written to disk, we can then prefetch the data in the third bucket into this same buffer.

The out-of-core algorithm appears in Figure 3. The only detail that is not provided is how to determine (k_1, k_2), the number of buckets in each table. We notice that it is better to have a small k_2. Each bucket in $M(u, it)$ requires two I/O operations while a bucket in the $M(u, it-1)$ table requires only one I/O operation. Consequently, we pick the smallest k_2 such that we have sufficient memory for the buffers and such that k_1 is less than the maximum number of open files. This lets us order the edges using a standard bucket sort

To conclude, we have presented two algorithms. The first operates entirely in-core and satisfies most of the requirements for our approximation algorithm. We further extended this algorithm to work as an external algorithm which needs little memory. We have argued that this algorithm satisfies all of the requirements that we set out in the introduction. We will now turn out attention to confirming these arguments with experimental evidence.

4 Experiments

In this section we present an experimental validation of our approach. We begin by presenting the data sets that will be used in the evaluation. Then, we introduce the existing algorithms that might be useful in computing the neighbourhood function. Next, because we are generating an approximation of a function over a potentially large domain, we propose a metric to evaluate the quality of this approximation. Once we have established this framework, a series of experiments will then be used to answer the following questions:

- Which approximation scheme provides the best results?
- How many parallel trials are needed for reasonable accuracy?
- Is sampling a reasonable way to compute the neighbourhood function?
- Is our algorithm sensitive to its parameters?
- How fast is the approximation?
- How well does the performance scale?

```
Select (k1, k2) to partition the two tables into buckets.

Partition the edges into their buckets, sorted first by their
second bucket and then to break ties by their first bucket.

FOR each node, u DO
   M(u,it-1) = M(u,it) = concatenation of k bitmasks, each with 1 bit set
                         (according to an exponential distribution)
   IF   we have filled up the buffer for M(u,it-1)
   THEN Asynchronously write M(u,it-1) and continue in the double buffer
   IF   we have filled up the buffer for M(u,it)
   THEN Asynchronously write M(u,it) and continue in the double buffer
DONE
Flush any buffers that need to be written

FOR each distance i DO
   Start fetching the first hunk of each table
   FOR each edge (u,v) DO
      IF M(u,it-1) is not in memory THEN
          We have been prefetching it, wait for it to be available
          Begin prefetching the next hunk of this table
      FI
      IF M(v, it) is not in memory THEN
          We have been writing out the last hunk (if applicable) and
          been prefetching the next hunk, wait for it to be available.

          Begin writing the past hunk (if applicable) and then
                prefetch the next hunk (if applicable)
      FI
      M(u,it) = (M(u, it) OR M(u, it-1))
   DONE

   /* Copy M(u,it) to M(u,it-1) as we stream through M(u,it) to compute the estimate */

   est = 0
   Start fetching the first hunk of M(u,it)
   FOR each node u DO
       IF M(u,it) is not in memory THEN
           We have been prefetching M(u,it) and now wait for it to be ready
       FI
       M(u,it-1) = M(u,it)
       If u marks the end of a hunk in M(u,it-1) THEN
           Start writing this hunk and schedule the prefetching of the next
           bucket to occur after this data has been written (if applicable)

           Continue processing in the double buffer
       FI
       est += (2^b)/(.7731*bias)                                          (*)
              b is the average of the least zero bits in the k bitmasks   (*)
   DONE
DONE
```

Figure 3: Approximation Neighbourhood Algorithm (ANF)

Name	#Nodes	#Edges	Degree Max.	Degree Avg.	Effective Diameter	Orient.	Real?
Cornell	844	1,647	131	1.95	9	Dir	Yes
Cycle	1,000	1,000	2	2.00	500	Undir	No
Grid	10,000	19,800	4	1.98	100	Undir	No
Uniform	65,378	199,996	20	3.06	8	Undir	No
Cora	127,083	330,198	457	2.60	35	Dir	Yes
80-20	166,946	449,832	723	2.69	10	Undir	No
Router	284,805	430,342	1,978	1.51	13	Undir	Yes

Table 2: Data set characteristics

4.1 Framework

4.1.1 Experimental Data Sets

We have collected three real data sets that cover many of the potential applications of the neighbourhood function. They are:

- **Router**: Undirected Internet routers data. The Internet Mapping project at Lucent Bell Laboratories [9] maps the Internet using *traceroute* and the SCAN project at ISI maps the Internet using distributed robots. We use the ISI merging of these two data sets which represent the current best map of the Internet (at a router level) [14].
- **Cornell**: Mark Craven collected crawls of various domains of the Web for use in web page classification. We use his crawl of the Cornell Web site.
- **Cora**: The CORA project at JustResearch crawled the Web, found Computer Science papers and extracted the meta information and citations [5]. We treat the citations as a directed graph where each node is a paper and each edge is a citation (from the citing paper to the cited paper).

We have selected four synthetic data sets. They are:

- **Cycle**: Graph that is a single simple undirected cycle (circle).
- **Grid**: Graph is a 2D planar grid (undirected).
- **Uniform**: Undirected graph in which edges are placed at random.
- **80-20**: Very skewed graph generated in an Internet like fashion with undirected edges [13].

Table 2 provides more detailed information about the graph properties. *Eff. diameter* is used informally to mean the smallest distance that includes most pairs of nodes. This table shows the coverage that we have achieved over different parameters.

Our synthetic data sets have been selected to increase coverage for some columns in this table and to test specific types of graphs. For example, the *Cycle* and *Grid* data sets have special forms that may be hard to approximate. We expect that these graphs of low maximum degree might be quite different from other graphs with exponentially related degrees. The 80-20 graph generation algorithm is used because it generates graphs that are similar to those found in the Internet and Web and it allowed us to test an intermediate sized "Internet" graph. Finally, a random graph was generated where there was a uniform probability of placing an edge between any pair of nodes.

4.1.2 RI-approximation scheme

The RI-approximation algorithm is based on the RI approximate counting scheme that we outlined in section 3.1. That is, the number of distinct elements in a multi-set is estimated by assigning a random value to each possible element and recording the least value that we actually saw.

To compute the neighbourhood function, we want to know for each node u the minimum value v_h of a node reachable from u in h hops, for $h = 1, 2, ...$; the estimate for the size of $S(u, h)$ is $\frac{1}{v_h} - 1$. The algorithm does the following equivalent, but more efficient procedure. Start from the node y with minimum value, and perform a breadth-first search from y following edges in reverse. If the BFS encounters u after h hops, we know that the value of y will be the minimum value among all nodes reachable from u in h or more hops, because it is the minimum of all values. We discard y from the graph and repeat with the new minimum, and so on for all the nodes. It was shown that this procedure takes only $O(m \log n)$ time with high probability, where m is the number of edges and n is the number of nodes in the graph. To reduce the variance in the estimates, the entire algorithm is repeated, averaging over the estimates.

4.1.3 Sampling

There are three possible ways to use sampling to approximate the neighbourhood function. The first two involve sampling the graph and the last involves sampling the neighbourhood computation.

First, we could take a random sample of the nodes of the graph and then use the subgraph that these nodes induce. That is, keep the edges for which both end-points are in our sample. The second approach is similar except we now sample the edges and keep the nodes that are in one or more edges. Neither of these methods is expected to be useful because the sampled graph is very

unlikely to resemble the real graph (imagine sampling a cycle, you will end up with disconnected arcs unless you take all nodes/edges). We verified that these methods are not practical and will consider them no further.

The last approach is much more compelling. It is akin to the sampling done in [11]. Recall that the neighbourhood function is:

$$N(h) = \sum_{u \in V} |S(u, h)|$$

Instead of taking the sum over all nodes, u, we could consider only a sample of the nodes. This has the potential to provide great estimates (consider again a graph that is a simple cycle – selecting any single point will give you a perfect estimate). However, we will experimentally show that this approach is not appealing for two reasons. First, because it lacks any error guarantees (we will construct a graph that has very large errors). Second, because it does not scale to the case that the edge file does not fits in memory. In addition to these two problems, it is also not possible to estimate the individual neighbourhood function $|S(u, h)|$.

4.1.4 Evaluation Metric

Our algorithm approximates a function that is defined over d points. To succinctly present our results, we need to define a metric which captures the quality of this estimation over all points. It is easy to define the quality of an estimation at a single point, h, as the relative error between the true function, $N(h)$, and the candidate approximation, $\hat{N}(h)$:

$$rel(N(h), \hat{N}(h)) = \frac{|N(h) - \hat{N}(h)|}{N(h)}$$

to combine the relative errors of all d points, we adopt the Root Mean Square (RMS) formula. That is, for a graph with diameter d, true neighbourhood function $N(h)$ and candidate approximate function $\hat{N}(h)$ we define

$$e(N, \hat{N}) = \sqrt{\frac{\sum_{h=2}^{d} rel(N(h), \hat{N}(h))^2}{d - 1}}$$

One thing to notice is that we begin the summation at $h = 2$. This is done since any algorithm may trivially compute the true value for $N(1)$ based on the number of nodes and edges in the graph. Instead of RMS, we could have simply computed the average error over all points. This was not done since the RMS metric further penalizes functions which have some very bad approximations. That is, we do not want a 5% error when half the points are perfect and half of them have a 10% error. In this case, the RMS error would be higher (7%). This metric captures the general "goodness" of the approximations.

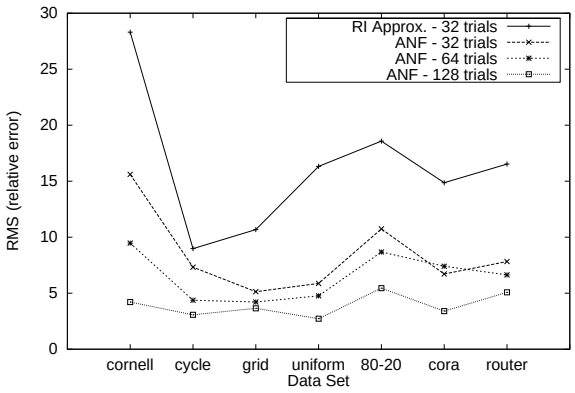


Figure 4: Our ANF algorithm provides much more accurate results than the RI approximation

4.2 Results

4.2.1 Accuracy

Perhaps the most important experiment is to see how accurate an approximation we can achieve on our data sets and to see if we have improved on the quality of the best previous approximation scheme. It turns out that we do get excellent approximations which are better and can be computed much faster than those from the RI-approximation scheme.

To see this, we conduct the following experiment. With fix the number of extra bits at $r = 7$ (in section 4.2.3 we will see that the results are relatively insensitive to this value) for our ANF algorithm. We then run our approximation 10 times for each data set. For each of these runs, we compute the RMS relative error and then average these 10 error values. As we see in Figure 4, for $k = 32$ approximations, our scheme has an error of at most 15%, for $k = 64$ that drops to at most 10% and finally at $k = 128$ the error is about 5% or less. We consider 10% to be a reasonable error and would recommend using the value $k = 64$ for most applications.

Thus, we can conclude that our approximation scheme provides excellent accuracy. Next, we want to know how this accuracy compares against the existing method of approximating the neighbourhood function. To do this, we will compare our approximations against the RI approximation generated with 32 trials. The same procedure was used to generate these numbers. We see that our approximations are much more accurate than those generated by the RI approximation scheme. Table 3 shows the time it takes to generate each of these approximations for the four largest data sets. We see that the 32 trials for the RI approximation is taking between one and two orders of magnitude longer to calculate than our $k = 128$ trial estimate.

Thus we can conclude that our approximation method provides less than a 10% error for $k = 64$ trials and that this is both much faster and much more accurate than the RI approximation scheme.

Method	Data Set			
	Uniform	Cora	80-20	Router
RI (32 trials)	20	98	241	941
ANF (k=128)	0.9	3.25	2.5	5.25
ANF (k=64)	0.5	1.5	1.5	2.75
ANF (k=32)	0.25	1	0.8	1.5

Table 3: Wall-clock running time (minutes) shows our ANF algorithm is much faster than the RI approximation

4.2.2 Sampling

As we explained in section 4.1.3, there are three problems with our sampling approach. They are:

- **Strong memory requirements**
  The breadth-first search procedure requires that the edges be held in memory. The entire edge file is accessed in a random order and using external storage for this task is too expensive. In [3, 10], they require a 12 GB machine with a very special data base to handle the edge file for the Web. This experimental framework is not available to most researchers.

- **Dependent on the graph**
  There are no bounds on the quality of the estimation. For a given graph, it is hard to know whether or not the estimate is reliable. As we will demonstrate, some graphs have huge errors (around 25% *relative* error even when sampling 15% of the possible start nodes).

- **No individual neighbourhood estimates**
  Both our ANF and the RI approximation provide estimates for the neighbourhood function of each node. This is not possible with the breadth-first sampling.

We now experimentally verify our intuition for the first two problems. To demonstrate these problems, we introduce a parameterized graph. Figure 5 helps illustrate the construction. First, create a chain of $d-2$ nodes that start from a node r and end at a node x. Add N nodes to the center of the graph, each of which has a directed edge to r and a directed edge from x. This gives a graph that has diameter d and a neighbourhood function that is $O(N)$ for each distance less than d and $O(N^2)$ for distance d. Finally, define a set of s source nodes that have an edge to each of the N center nodes and a set of t terminal nodes that have an edge from each of the N center nodes. If $N >> s$ and t then the majority of the sampled nodes will be from the N center nodes and few will be from the s start nodes. This will result in an error which is a factor of around s/p for a p percent sampled search.

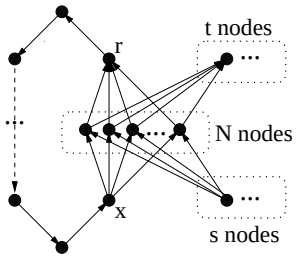
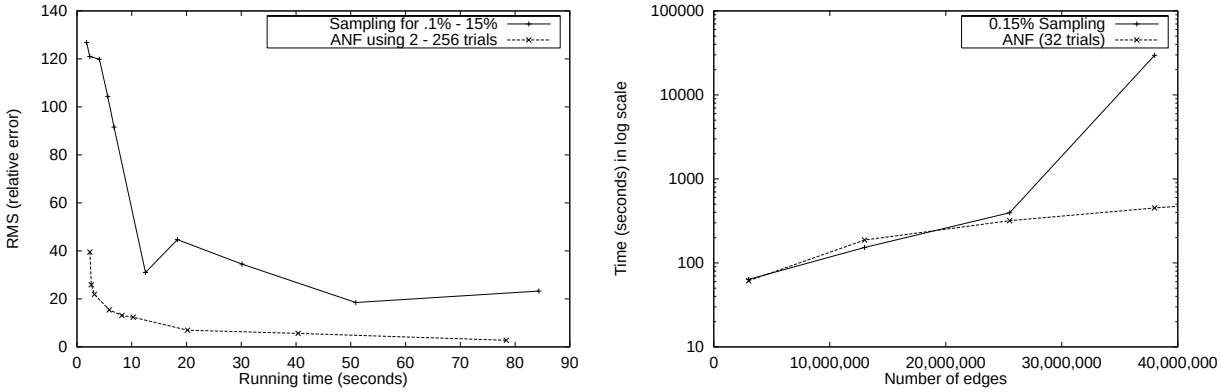


Figure 5: Simple graph that shows poor behaviour for sampling



a) Error vs. Running time b) Running time vs. Number of edges

Figure 6: Sampled breadth-first search can provide huge errors and does not scale to very large edge files

We see this large error in Figure 6 a). For sample sizes between .1% and 15%, we ran the sampled breadth-first approach and measured the wall-clock time and the RMS (relative error). These values are then plotted as a scatter-plot and a line has been added to illustrate the trends as the sample size grows. For comparison, we used the same procedure for our ANF approach and varied the number of parallel trials between 2 and 256. It is easy to see that the sampling approach never produces a useful result and that our approximation both runs faster and quickly converges to a small error. The graph was generated with $N = 25,000$, $s = 100$, $t = 100$ and $d = 6$ and the results are averaged over 20 trials.

The second problem that we identified was that breadth-first sampling makes random accesses into the edge file. To illustrate this problem, begin with a graph that has $N = 250,000$ and $s = t = 5$. We then increase s, t until the edge file no longer fits in memory and measure the running time of sampling and ANF. We use a 0.15% sample and use $k = 32$ parallel trials for ANF. This provides approximately the same running time for the first point on the graph. In Figure 6 b) we see that both algorithms scale linearly except that there is a *huge* (approximately 100x increase)

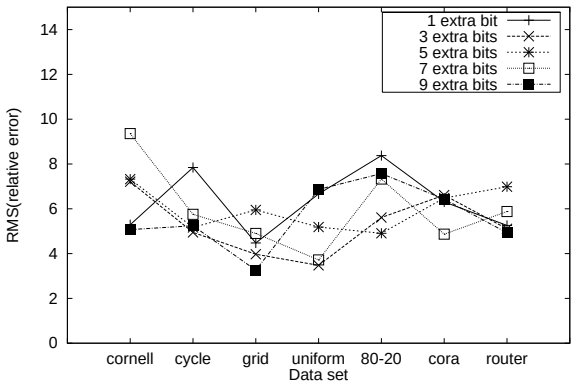


Figure 7: Results are not sensitive to the value of parameter r

jump in running time for the breadth-first search when the edge file is larger than the memory.

For these reasons, we conclude that sampled breadth-first search is not a viable solution.

4.2.3 Parameter Sensitivity

Our algorithm has only two parameters. The first is the number of random trials to use. In section 4.2.1, we measured the quality of the approximation for various values of this parameters. For these experiments, we fix $k = 64$ as this is the most likely value to be used in practice. We then vary the second parameter, the number of additional bits to use (r). Using all our data sets, we pick various values for r and measure the resulting RMS error over 10 trials. These results appear in Figure 7. We find that the accuracy is not very sensitive to the value of r and that a value of r of around 5 or 7 provides consistent results.

4.2.4 Alternatives

We have also considered some alternatives in our ANF algorithm. In Figures 1 and 3 there are two lines marked with (*). At this point in the algorithm, we are taking the average of all the individual bit estimates over the k parallel trials. It is interesting to consider other alternatives here. We experimentally attempted several and found none to be obviously superior to the averaging method that we are presenting. The different methods that we considered are:

- **Median** Take the median of the k values $b_1, \cdots, b_k$.
- **Average** The method in the algorithm.
- **Median-3** Partition the k values in three groups b_1, $\cdots$, $b_{k/3}$, $b_{k/3+1}$, $\cdots$, $b_{2k/3}$ and $b_{2k/3+1}$, $\cdots$, b_k. Compute the average b in each of these sets and then take the median of the three

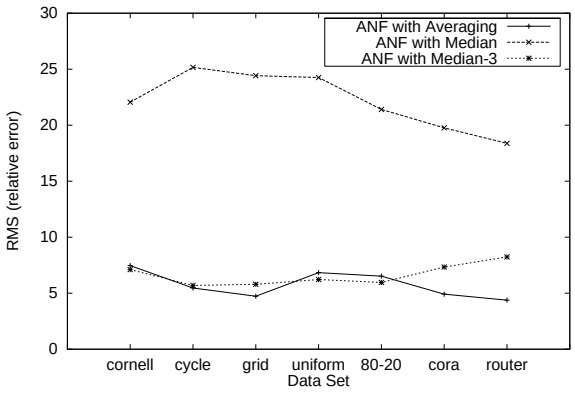


Figure 8: Errors using different methods of combining the k trials

averages.

The first two methods are obvious to try. The last one was considered because it may make the results less sensitive to a gross over or under estimate. To test these methods, we ran a typical experiment. We fixed $k = 64$ trials and varied the averaging method in the algorithm. For each data set, we ran 10 trials and averaged the RMS error of each trial. These results appear in Figure 8. Here we see that *Median* is clearly worse. Among the other two options, there is no clearly superior choice.

From this experiment, we conclude that our averaging approach is good and that taking the median is a poor choice.

4.2.5 Speed

Table 4 reports will-clock running times for the exact computation (Breadth-First search) compared to the running times for our approximation scheme (ANF). The times were collected by averaging over several runs on an otherwise unloaded Pentium II-450 machine with 256 MB of RAM. In all cases, there was sufficient memory to process the entire data set in memory. The approximations use 64 random trials which, for our approximation, has an expected error of less than 10% for all data sets. We chose to use this number of approximations because it seems to be an appropriate choice for most applications: ensuring a high degree of accuracy as well as very fast execution.

From this, we can conclude that the approximations are very fast. For the *Router* data set, the running time is reduced from close to a day to a couple of minutes. The *Cora* data set are surprising. Upon further investigation, It turns out that the majority of the nodes in this graph have 0 in-degree. In retrospect, this is a sad truth of publishing: the majority of the papers written are never cited. Effectively, we are running the breadth-first search on a large number of

Data Set	BF (Exact)	ANF	Speed-up
Uniform	92	0.5	184x
Cora	6	1.5	4x
80-20	680	1.5	453x
Router	1,200	2.75	436x

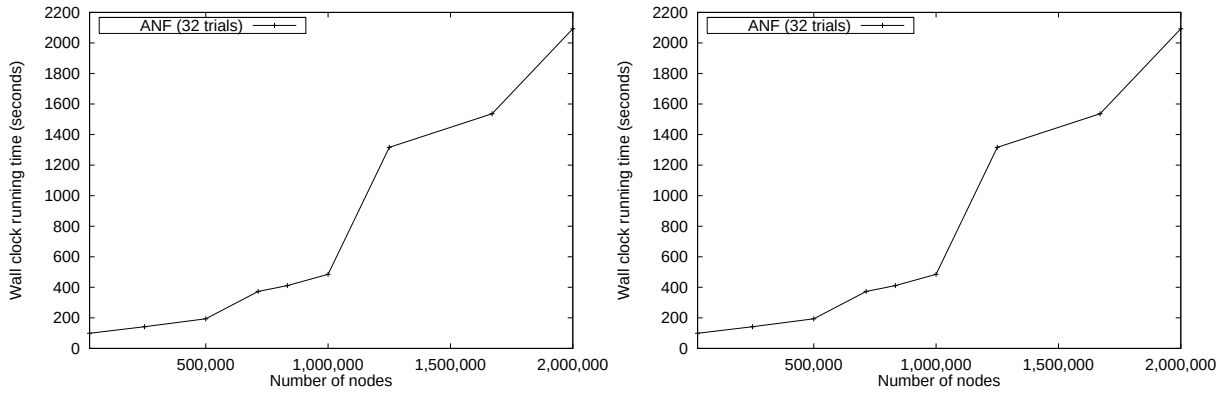
Table 4: Wall clock running time (minutes) and huge speed-up for *ANF*

very small graphs, which provides the surprisingly fast time to compute the exact neighbourhood function. However, even in a graph that is skewed so heavily to favour the running time of the exact computation, we see a reasonable speed-up when running our ANF approximation. From this table we can conclude that huge speed-ups are possible (especially as the graphs get larger, such as Web graphs) and that even in really unfortunate situations, we are still significantly faster than the exact computation.

To extrapolate from these times, we conducted two additional experiments. When looking at the performance of sampling approaches, we introduced a parameterized graph (see Figure 5). This graph makes it easy to control the number of nodes and edges while ensuring that we can compute the true neighbourhood function explicitly. For these reasons, we continue to use this graph to examine how the running time of our approximation scales (in terms of the number of nodes and the number of edges).

Figure 9 shows the change in running time as the number of nodes increases (while holding the number of edges essentially fixed). To do this, we vary N and pick s and t such that the resulting graph has approximately the same number of edges (see Figure 5). For various numbers of nodes, we generate the graph and run 3 trials on a Pentium II-400 machine with 128MB of RAM to compute the average wall-clock running time (in seconds). We have used this machine because it has relatively little memory which allows us to measure the time as our memory requirements exceed the amount of available memory. Figure 9 has three distinct segments:

- From 0 to 500,000: all data fits into memory and there is only a linear increase in running time (with a small constant).
- Between 500,000 and 1,000,000 nodes: at least half of the data that we use fits into memory. There is a slight jump in running time but otherwise it scales linearly (with the same small constant).
- More than 1,000,000 nodes: another jump in the running time but again the processing scales linearly.



a) Scales with number of nodes b) Scales with number of edges

Figure 9: Running time scales linearly with the number of nodes and edges

This allows us to conclude that our algorithm scales *very* well with the number of nodes.

A similar experiment was conducted for the running time as the number of edges increase. Again, we used the parameterized graph from Figure 5. This time, we increase s and t to increases the number of edges without greatly changing the number of nodes. Here we see a constant linear increase in running time (measuring up to 50,000,000 edges).

5 Applications

In the introduction, we outlined some of the applications which currently make use of a neighbourhood function. We have found many new applications. We will look at ways to mine the contents of large graphs, to mine collections of large graphs and other graph problems that we have also solved. This is not intended to be an exhaustive treatment of these points, but merely an indication of the directions which are now available to us because it is possible to approximate the neighbourhood function so quickly.

Connected Components In addition to the neighbourhood function, we have a way to determine the strongly connected components of the graph. Upon convergence, we have computed a set of random bitmasks for each node. If two nodes are in the same strongly connected component, they will have exactly the same bitmasks. If they lie in different strongly connected components, there is a very high probability that they will have at least one pair of bitmasks that differ. Thus as we compute the approximate neighbourhood function, we can also compute the connected components. If we know the set of connected components, we can actually compute the true value of

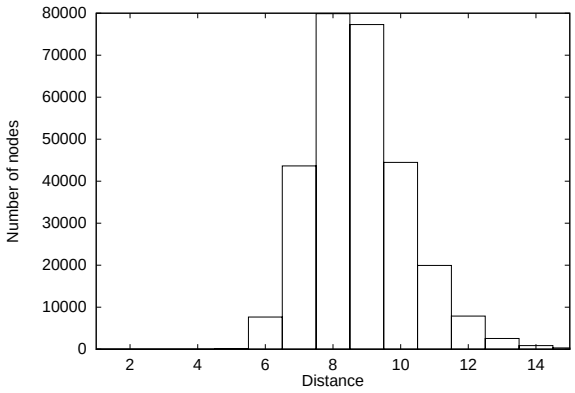


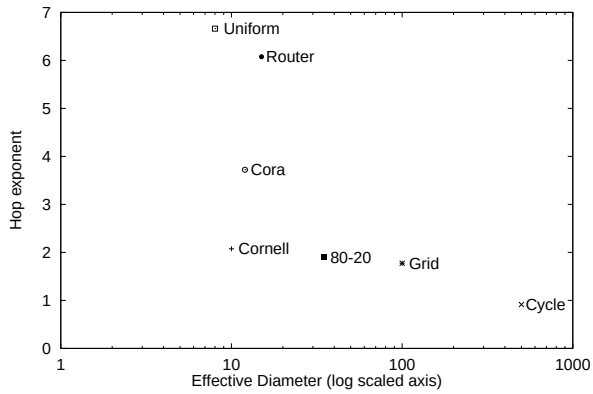
Figure 10: Histogram of the minimum distances needed to be able to reach at least half of the nodes in the *Router* data set

$N(\infty)$ exactly. Given the true value $N(\infty)$ and the approximation $\hat{N}(\infty)$, we have some information about the quality of a given approximation. That is, we know the relative error in the limit which gives us some assurances about the relative quality of the approximation.

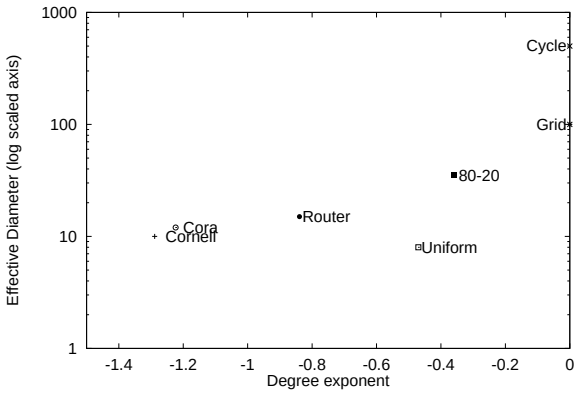
Mining the Web In [3, 10], the authors discuss the Web as a large graph. To think of the so-called *small world phenomenon* on the Web (where they find that 75% of web pages do not have a path between them), they use the average distance for all pairs that are in the same weakly connected component. We can also compute the average distance with our approximation. Since there are $N(h)$ pairs at distance h or less and $N(h-1)$ at distance $h-1$ or less, there are exactly $N(h) - N(h-1)$ that are distance h apart. Their analysis would have been helped by the use of our approximation.

Discriminating among nodes We can think of other approaches to mining large graphs that might be profitable. It is useful to know if pairs of nodes are near or far apart. To do this, we ran the following experiment. For each node, u, compute the minimum d_u for which $|S(u, d_u)| \geq n/2$. That is, the smallest distance at which u can reach half of the graph. Figure 10 shows a histogram of these distances. Here we see that most nodes take 7 or 8 hops to reach most of the graph which suggests that the backbone of the Internet is around 6 hops. Furthermore, we see that 35,000 nodes are much further away from the bulk of the Internet than all others.

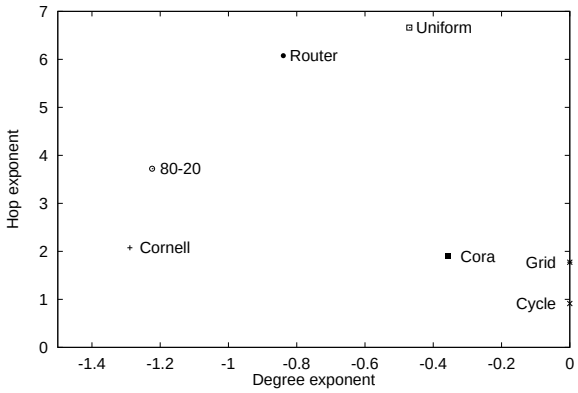
New ideas for indexing and mining large graphs Recent research into the nature of the Internet and the Web has discovered that power-law relationships seem to hold [6, 3, 10, 2]. Two interesting power-laws have to do with the out-degree and with the neighbourhood function. The



(a) Hop exponent vs. Diameter



(b) Diameter vs. Degree exponent



(c) Hop exponent vs. Degree exponent

Figure 11: Pair-wise combinations of the three features extracted from the graphs

former is defined as follows. Compute the out-degree of each node in the graph and sort them. The out-degree of a node is then inversely exponentially proportional to its rank. We call the exponent of this relationship the *degree exponent*. The latter power-law says that there is an exponential relationship between the number of hops and the neighbourhood size. We call this exponent the *hop exponent*. These two exponents are features that we can extract from the graph. A third potentially useful feature is the effective diameter (or diameter). These three features provide useful ways of indexing a collection of large graphs by storing them in R-Trees and offering similarity search on the set of graphs. We look at the 3 pairwise combinations of the attributes in Figure 11. For example, using the *degree exponent* and the *hop exponent* we find that the two very synthetic graphs (*Grid* and *Cycle*) are the most similar and all other graphs are further apart. The three graphs suggest that our features have potential value in mining and indexing large graphs. In particular, the Internet Mapping Project at Lucent has time indexed graphs of the Internet. They have seen major network differences during the bombing of Yugoslavia and when major network

problems occur. Using these features, it may be possible to visualize the set of all these graphs to look for days in which interesting changes are happening.

6 Conclusions

We have presented a novel algorithm for computing an approximation of the neighbourhood function. We found both analytically and experimentally that it:

- **Provides highly-accurate estimates:** Provable bounds and experimentally we see less than 10% error for $k = 64$ trials for all our synthetic and real data sets.
- **Is orders of magnitude faster:** On the seven data sets used in this paper, our algorithm is up to two orders of magnitude faster than the exact computation. It is also much faster than the only existing approximation scheme.
- **Has low storage requirements:** Given the edge file, our algorithm uses only $O(n)$ additional storage.
- **Adapts to the available memory:** We presented an out-of-core version of our algorithm and experimentally verified that it scales smoothly with the graph size.
- **May be parallelized:** Our ANF algorithm may be the parallelized with very few synchronization points.
- **Employs sequential scans:** Unlike previous approaches for approximating the neighbourhood function, our algorithm avoids random access of the edge file and performs one sequential scan of the edge file per hop.
- **Estimates individual neighbourhood functions for free:** As a byproduct of our algorithm, we obtained estimates for $|S(u, h)|$ for all u and h. These estimates have provable guarantees on their accuracy, regardless of the graph, and are useful in classifying nodes, determining connected components, etc.

Because it is now possible to compute accurate estimates for the neighbourhood function, we found new and interesting ways in which it could be used. By looking at the estimates of the individual functions, $|S(u, h)|$, we found that we could gain insight into the structure of the graph. By using power-law exponents, we found that we could select graph features that allow us to index large graphs using R-trees and allows us to perform similarity search on their structure. All of these

applications are new ways of mining information from large graphs or collections of large graphs that are enabled as a result of the speed of our approximate neighbourhood computation.

References

[1] L. A. Adamic. The small world Web. In *Proceedings of the European Conf. on Digital Libraries*, 1999.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, and R. Stata. Graph structure in the Web. In *Proceedings of the 9th International World Wide Web Conference*, pages 247–256, 2000.

[4] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, December 1997.

[5] CORA search engine. `http://www.cora.whizbang.com`.

[6] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, 1999.

[7] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[8] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 323–334, 1995.

[9] `http://cm.bell-labs.com/who/ches/map/index.html`.

[10] S. R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. The Web as a graph. In *ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems*, pages 1–10, 2000.

[11] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Proceedings of 15th International Conference on Very Large Data Bases*, pages 315–326, 1989.

[12] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of 25rd International Conference on Very Large Data Bases*, pages 315–326, 1999.

[13] C. R. Palmer and J. G. Steffan. Generating network toplogies that obey power laws. In *IEEE Globecom 2000*, 2000.

[14] `http://www.isi.edu/scan/mercator/maps.html`.

This research was sponsored in part by National Science Foundation (NSF) grant no. CCR-0122581.