ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs

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

Phillip B. Gibbons
Intel Research Pittsburgh
Pittsburgh, PA
phillip.b.gibbons@intel.com

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

ABSTRACT

Graphs are an increasingly important data source, with such important graphs as the Internet and the Web. Other familiar graphs include CAD circuits, phone records, gene sequences, city streets, social networks and academic citations. Any kind of relationship, such as actors appearing in movies, can be represented as a graph. This work presents a data mining tool, called ANF, that can quickly answer a number of interesting questions on graph-represented data, such as the following. How robust is the Internet to failures? What are the most influential database papers? Are there gender differences in movie appearance patterns? At its core, ANF is based on a fast and memory-efficient approach for approximating the complete "neighbourhood function" for a graph. For the Internet graph (268K nodes), ANF's highly-accurate approximation is more than **700 times faster** than the exact computation. This reduces the running time from nearly **a day to a matter of a minute or two**, allowing users to perform ad hoc drill-down tasks and to repeatedly answer questions about changing data sources. To enable this drill-down, ANF employs new techniques for approximating neighbourhood-type functions for graphs with distinguished nodes and/or edges. When compared to the best existing approximation, ANF's approach is both faster and more accurate, given the same resources. Additionally, unlike previous approaches, ANF scales gracefully to handle disk resident graphs. Finally, we present some of our results from mining large graphs using ANF.

1. INTRODUCTION

Graph-based data is becoming more prevalent and interesting to the data mining community, for example in understanding the Internet and the WWW. These entities are modeled as graphs where each node is a computer, administrative domain of the Internet, or a web page, and each edge is a connection (network or hyperlink) between the resources. Google is interested in finding the most "important" nodes in such a graph [2]. Broder et al. studied the connectivity information of nodes in the Internet [13, 3]. The networking community has used different measures of node "importance" to build a hierarchy of the Internet [20]. Another source of graph data that has been studied are citation graphs [18]. Here, each node is a publication and each edge is a citation from one publication to another and we wish to know the most important papers. There are many more examples of graphs which contain interesting information for data mining purposes. For example, the telephone calling records from a long distance carrier can be viewed as a graph, and by mining the graph we may help identify fraudulent behaviour or marketing opportunities. DNA sequencing can also be viewed as a graph, and identifying common subsequences is a form of mining that could help scientists. Circuit design, for example from a CAD system, forms a graph and data mining could be used to find commonly used components, points of failure, etc.

In fact, any binary relational table is a graph. For example, in this paper we use a graph derived from the Internet Movie Database [10] where we let each actor and each movie be a node and add an undirected edges between and actor, a, and a movie, m, to indicate that a appeared in m. It is also common to define graphs for board positions in a game. We will consider the simple game of tic-tac-toe. From all of these data sources, we find some prototypical questions which have motivated this work:

1. How robust is the Internet to failures?
2. Is the Canadian Internet similar to the French?
3. Does the Internet have a hierarchical structure?
4. Are phone call patterns (caller-callee) in Asia similar to those in the U.S.?
5. Does a new circuit design appear similar to a previously patented design?
6. What are the most influential database papers?
7. Which set of street closures would least affect traffic?
8. What is the best opening move in tic-tac-toe?
9. Are there gender differences in movie appearances?
10. Cluster movie genres.

It is possible to answer all of these questions by computing three graph properties pertaining to the connectivity or neighbourhood structure of the graph:

Graph Similarity: Given two graphs, G_1 and G_2, do the graphs have similar connectivity / neighbourhood structure

(independent of their sizes). Such a similarity measure is useful for answering questions 1, 4, and 5.
Subgraph Similarity: Given two subsets of the vertices of the graph, V_1 and V_2, compare how these two induced subgraphs are connected within the graph. Such a similarity measure is useful for answering questions 2, 4, 8, 9, and 10.
Vertex Importance: Assign an importance to each node in the graph based on the connectivity. This importance measure is useful for answering questions 1, 3, 6, and 7.

We answer questions 1, 7 and 10 in this paper, one from each of the three types. The remaining questions can be answered in a similar fashion, using various forms of the *Neighbourhood Function*. The basic neighbourhood function, $N(h)$, for a graph, also called the *hop plot* [8], is the number of pairs of nodes within a specified distance h, for all distances h. In section 2 we will define this more formally and present a more general form of the neighbourhood function that can be used to compute all three graph properties.

The main contribution of this paper is a tool that allows us to compute these three graph properties, thereby enabling us to answer interesting questions like those we suggested. Beyond simply answering the questions, we want our tool to be fast enough to allow drill-down tasks. That is, we want it to be possible to interactively answer users requests. For example, to determine the best roads to close for a parade, the city planner would want to interactively consider various sets of street closures and compare the effect on traffic. Almost in contrast to the need to be able to run interactively on graphs, we also want a tool that scales to very large graphs. In [3, 13], measuring properties about the web required hardware resources that are beyond the means of most researchers. Instead, we produce a data mining tool that is useful given any amount of RAM. These two goals give rise to the following list of properties that our tool must satisfy when analyzing a graph with n nodes and m edges:

Error guarantees: estimates must be accurate at all distances (not just in the limit).
Fast: scale linearly with # of nodes and # edges (n, m).
Low storage requirements: use only $O(n)$ additional storage.
Adapts to the available memory: when the node set does not fit in memory, make effective use of the available memory.
Parallelizable: for massive graphs, must be able to distribute the work over multiple processors and/or multiple workstations, with low overheads.
Sequential scans of the edge file: avoid random accesses to the graph. Random accesses exhibit horrible paging performance for the common case that the graph is larger than the available memory.
Estimates per node: must be able to estimate the neighbourhood function from each node, not just for the graph as a whole.

This paper presents such a tool, which we call ANF for *Approximate Neighbourhood Function*. In the literature, we have found two existing approaches that could prove useful for computing the neighbourhood function. We show that neither meets our requirements, primarily because neither scales well to very large graphs. This can be seen in Figure 1,

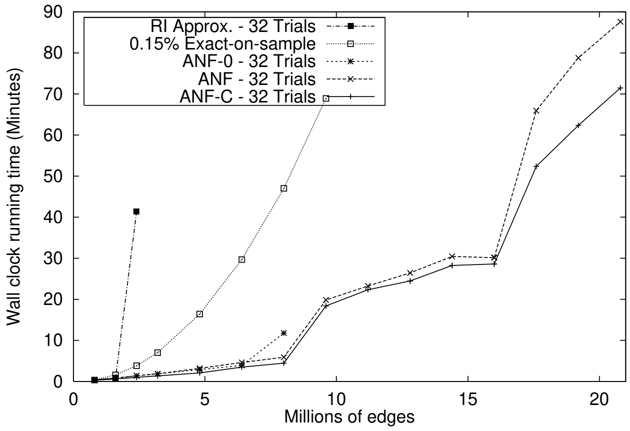


Figure 1: ANF algorithms scale but not the others

which plots the running time versus the graph size for some randomly-generated graphs. The two existing approaches (the RI approximation scheme [4] and a random sampling approach) scale very poorly while our ANF schemes scale much more gracefully and make it possible to investigate much larger graphs. In section 2 we provide background material, definitions and a survey of the related work. In section 3 we describe our ANF algorithms. In section 4, we present experimental results demonstrating the scalability of our approach. In addition, we show that, given the same resources, (1) ANF is much more accurate and faster then the RI approximation scheme, and (2) ANF is more than 700 times faster than the exact computation for a snapshot of the Internet graph (268K nodes). In section 5, we use ANF to answer some of the prototypical questions posed in this introduction.

2. BACKGROUND AND RELATED WORK

2.1 Definitions

Let $G = (V, E)$ be a graph with n vertices, V, and m directed edges, E. (Table 1 summarizes the symbols used in this paper.) Let $dist(u, v)$ be the number of edges on the shortest path from u to v. To answer our prototypical questions, we need a characterization of a node's connectivity and the connectivity within a graph, as a whole. Accordingly, we define the following forms of the neighbourhood function:

Def. 1. The *individual neighbourhood function* for u at h is the number of nodes at distance h or less from u.
$IN(u, h) = |\ \{v : v \in V, dist(u, v) \le h\}\ |$
Def. 2. The *neighbourhood function* at h is the number of pairs of nodes within distance h.
$N(h) = |\ \{(u, v) : u \in V, v \in V, dist(u, v) \le h\}\ |$, or
$N(h) = \sum_{u \in V} IN(u, h)$.

To deal with subgraphs, we generalize these two definitions slightly. Let S be a set of *starting nodes* and C be a set of *concluding nodes*. We are interested in the number of pairs starting from a node in S to a node in C within distance h:

Def. 3. The *generalized individual neighbourhood function* for u at h given C is the number of nodes in C within distance h.
$IN^+(u, h, C) = |\ \{v : v \in C, dist(u, v) \le h\}\ |$.
Note that $IN(u, h) = IN^+(u, h, V)$.

Table 1: Commonly used symbols

Name	Description
n	Number of vertices
m	Number of edges
V	Vertex set $\{0, 1, \cdots, n-1\}$
E	Edge set $\{(u, v) : u, v \in V\}$
d	Diameter of the graph
S	Starting set for the neighbourhood
C	Concluding set for the neighbourhood
r	Number of extra approximation bits
k	Number of parallel approximations

Def. 4. The *generalized neighbourhood function* at h is the number of pairs of a node from S and a node from C that are within distance h or less.

$N^+(h, S, C) = |\ \{(u, v) : u \in S, v \in C, dist(u, v) \leq h\}\ |$

$N^+(h, S, C) = \sum_{u \in S} IN^+(u, h, C)$.

Note that $N(h) = N^+(h, V, V)$.

In section 5 we will use the neighbourhood function to characterize graphs. We will compare $N_{G_1}(h)$ to $N_{G_2}(h)$ to measure the similarity in connectivity/neighbourhood structure of graphs G_1 and G_2. For example, if we want to know the structural similarly of the Web from 1999 to today's Web, we can compute their neighbourhood functions and compare them at all points. Comparing subgraphs induced by vertex sets V_1 and V_2 can be done by comparing $N^+(h, V_1, V_1)$ to $N(h, V_2, V_2)$. E.g., let V_1 be the routers in the Canadian domain and V_2 be the routers in the French domain. Finally, we will use the individual neighbourhood function for a node to characterize its importance, with respect to the connectivity. E.g., the most important router is the one that in some way reaches the most routers the fastest.

Thus, if we can compute all these variants of the neighbourhood function efficiently, then we can answer the graph questions that we posed in the introduction.

2.2 Related Work

It is trivial to compute $N(0)$ and $N(1)$, which are $|V|$ and $|V| + |E|$ respectively. $N(2)$ is reminiscent of the size of the (self-)join of the edge relation: each edge is viewed as a tuple with attributes "first" and "second" and $N(2) - N(1)$ is the size of the result of the query

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

Writing $N(2) - N(1)$ in this way illustrates the difficulty in efficiently computing $N(h)$ for any $h \geq 2$. The *distinct* means that we must know which of the n^2 possible pairs of nodes have already been counted and we must take care not to over count in the presence of multiple paths between the same pair of nodes. One approach to computing $N(h)$ is to repeatedly multiply the graph's adjacency matrix. Asymptotically, this could be done in $O(n^{2.38})$ time. Unfortunately, we would also require $O(n^2)$ memory, which is prohibitive. Instead, it is common to use breadth first searches in the graph. A breadth-first search beginning from u can easily compute $IN(u, h)$ for all h. We can compute $N(h)$ by running a breadth-first search from each node u and summing over all u. This takes only $O(n + m)$ storage but the worse case running time is $O(nm)$. For large, disk resident graphs, a breadth-first search results in an expensive random-like access to the disk blocks. This appears to be the state of the art solution for exact computation of $N(h)$.

The transitive closure is $N(\infty)$ or, equivalently, $N(d)$, where d is the diameter of the graph. Lipton and Naughton [14] presented an $O(n\sqrt{m})$ algorithm for estimating the transitive closure that uses an adaptive sampling approach for selecting starting nodes of breadth-first traversals. Motivated by this work, in section 4 we will experimentally evaluate a similar sampling strategy to discover that it scales poorly to large graphs and, due to the random-like access to the edge file, it does not scale to graphs larger than the RAM. Most importantly, however, we will find that the quality of this approximation can be quite poor (we show an example where even a sample as large as 15% does not provide a useful approximation!). Lipton and Naughton's work was improved by Cohen, who gave an $O(m)$ time algorithm using only $O(n + m)$ memory [4]. Cohen also presented an $O(m \log n)$ time algorithm for estimating the individual neighbourhood functions, $IN(u, h)$. This appears to be the only previous work which attempts to approximate the neighbourhood function. More details of this algorithm, which we refer to as the *RI approximation*, appear in section 4.1.1 when we experimentally compare it to our new approximation. Our experiments demonstrate that the RI approximation is ill-suited for large graphs; this is due to its extensive use of random-like access (for breadth first search, heap data structures, etc.).

The problems of random access to a disk resident edge file has been addressed in [15]. They find that it is possible to define good storage layouts for undirected graphs but that the storage blowup can be very large. Given that we are interested only in very large graphs and graphs with directed edges, this does not solve the problems related to large edge files. Instead, we will need to find a new computation strategy which avoids random access to disk.

State-of-the-art approaches to understanding/characterizing the Internet and the Web very often make use of neighbourhood information [3, 13, 1, 20]. Other recent work in data mining for graphs has focused on mining frequent substructures. Essentially, the problem of finding frequent itemsets is generalized to frequent subgraphs. Systems for this include SUBDUE [5] and AGM [11]. Graphs have been used to improve marketing strategies [7]. A survey of work on citation analysis appears in [18].

3. PROPOSED APPROXIMATION TO THE NEIGHBOURHOOD FUNCTION

We are given a graph $G = (V, E)$. We assume that $V = \{0, 1, \cdots, n-1\}$ and that E contains m directed edges. Undirected graphs can be represented using pairs of directed edges. We wish to approximate the function $N^+(h, S, C)$ and $IN^+(x, h, C)$ for a node x, allowing us to compute $N(h)$ and $IN(x, h)$. The approximation must be done accurately and in such a way that we will be able to handle disk resident graphs. In this section, we construct such an approximation gradually. First, we approximate $N(h)$ and/or $IN(x, h)$ assuming memory-resident data structures. We extend this algorithm to approximate $N^+(h, S, C)$ and $IN^+(x, h, C)$ but still requiring sufficient RAM for processing. Next, we move

// Set $\mathcal{M}(x,0) = \{x\}$
FOR each node x DO
    $M(x,0) =$ concatenation of k bitmasks
        each with 1 bit set ($P(\text{bit } i) = .5^{i+1}$)
FOR each distance h starting with 1 DO
    FOR each node x DO $M(x,h) = M(x,h-1)$
    // Update $\mathcal{M}(x,h)$ by adding one step
    FOR each edge (x,y) DO
        $M(x,h) = (M(x,h)$ BITWISE-OR $M(y,h-1))$
    // Compute the estimates for this h
    FOR each node x DO
        Individual estimate $I\hat{N}(x,h) = (2^b)/.77351$
        where b is the average position of the least zero bits
        in the k bitmasks
    The estimate is: $\hat{N}(h) = \sum_{\text{all x}} I\hat{N}(x,h)$

Figure 2: Introduction to the basic ANF algorithm

the data structure to disk and to create an algorithm that meets all of our requirements. Finally, we will extend this algorithm with bit compression to further increase its speed.

3.1 Basic ANF Algorithm (ANF-0)

A graph traversal effectively accesses the edge file in random order. Thus, if our algorithm is going to be efficient on a graph that does not fit in memory, we cannot perform any graph traversals. Instead, we are going to build up the nodes reachable from x within h steps by first finding out which nodes its neighbours can reach in $h-1$ steps. Slightly more formally, let $\mathcal{M}(x,h)$ be the set of nodes within distance h of x. Clearly, $\mathcal{M}(x,0) = \{x\}$, since the only node within distance 0 of x is x itself. To compute $\mathcal{M}(x,h)$ we note that x can still reach in h or fewer steps the nodes it could reach in $h-1$ or fewer steps. But, x can also reach the nodes in $\mathcal{M}(y,h-1)$ if there is an edge from x to y. That is:

$\mathcal{M}(x,0) = \{x\}$ for all $x \in V$
FOR each distance h DO
    $\mathcal{M}(x,h) = \mathcal{M}(x,h-1)$ for all $x \in V$
    FOR each edge (x,y) DO
        $\mathcal{M}(x,h) = \mathcal{M}(x,h) \cup \mathcal{M}(y,h-1)$

This iterates over the edge set instead of performing a traversal. The trick will be to efficiently compute the number of distinct elements in $\mathcal{M}(x,h)$. One possibility is to use a dictionary data structure (e.g., a B-tree) to represent the sets $\mathcal{M}(x,h)$. However, this approach needs $O(n^2 \log n)$ time and space, which is prohibitive. An approach that people, particularly C hackers, often employ is to use bits to mark membership. That is, each node is given one of n bits and a set is a bit string of length n. To add a node to the set, we mark its bit. The union of two sets is the bitwise-OR of the bitmasks. Unfortunately, this approach still uses $O(n^2)$ memory, which will be prohibitive for large graphs.

Instead, we're going to use a clever probabilistic counting algorithm [9] to approximate the sizes of the sets using shorter bit strings ($\log n + r$ bits, for some small constant r). We refer to the bit string that approximates $\mathcal{M}(x,h)$ as $M(x,h)$. Instead of giving each node its own bit, we are going to give about half the nodes bit 0, a quarter of them bit 1, and so on (give a node bit i with probability $1/2^{i+1}$). We still mark a node by setting its bit and use bitwise-OR for the set-union. Estimating the size of the set from the small bit string is done based on the following intuition. If we expect 25% of the nodes to be assigned to bit 1 and we haven't seen any of them (bit 1 is not set), then we probably saw about 4 or less nodes. So, the approximation of the size of the set $\mathcal{M}(x,h)$ is proportional to 2^b, where b is the least bit number in $M(x,h)$ that has not been set. We refer the reader to [9] for a derivation of the constant of proportionality and a proof that this estimate has good error bounds.

x	$M(x,0)$	$M(x,1)$	$I\hat{N}(x,1)$	$M(x,2)$	$I\hat{N}(x,3)$
0	100 100 001	110 110 101	4.1	110 111 101	5.2
1	010 100 100	110 101 101	3.25	110 111 101	5.2
2	100 001 100	110 101 100	3.25	110 111 101	5.2
3	100 100 100	100 111 100	4.1	110 111 101	5.2
4	100 010 100	100 110 101	3.25	110 111 101	5.2

Figure 3: Simple example of *basic ANF*

FOR each node x DO
    IF $x \in C$ THEN
        $Mcur(x) =$ concatenation of k bitmasks each
            with 1 bit set ($P(\text{bit } i) = .5^{i+1}$)
FOR each distance h starting with 1 DO
    FOR each node x DO $Mlast(x) = Mcur(x)$
    FOR each edge (x,y) DO
        $Mcur(x) = (Mcur(x)$ BITWISE-OR $Mlast(y))$
    FOR each node x DO
        $I\hat{N}^{+}(x,h,C) = (2^b)/.77351$, where b is the average
        position of the least zero bit in the k bitmasks
    $\hat{N}^{+}(h,S,C) = \sum_{x \in S} I\hat{N}(x,h,C)$

Figure 4: ANF-0: In-core ANF

A single approximation is obviously not very robust. We do k parallel approximations by treating $M(x,h)$ as a bitstring of length $k(\log n + r)$ bits. Figure 2 shows the complete algorithm implementing the edge-scan based ANF.

Example. Figure 3 shows the bitmasks and approximations for a simple example of our most basic ANF algorithm. The purpose is to clarify the concatenation of the bitmasks and to illustrate the computation. The input is a 5 node undirected cycle and we used parameters $k = 3$ and $r = 0$. The first `FOR` loop of the algorithms generates the table of random bitmasks $M(x,0)$. That is, using an exponential distribution, we randomly set one bit in each of the three concatenated bitmasks. (In the figure, bit 0 is the leftmost bit in each 3-bit mask.) Then, each iteration uses the `OR` operation to combine the nodes that it could reach in $h-1$ steps plus the ones that its neighbours could reach in $h-1$ steps. For example, $M(2,1)$ is just $M(1,1)$ `OR` $M(2,1)$ `OR` $M(3,1)$, because nodes 1 and 3 are the neighbors of node 2. The estimates, for example $I\hat{N}(2,1)$, are computed from the average of the least zero bit positions ($2,1,1 = \frac{4}{3}$, and $2^{4/3}/.77359 = 3.25$).

The algorithm in Figure 2 uses an excessive amount of memory and does not estimate the more general forms of the neighbourhood function. Figure 4 depicts the same algorithm, with the following improvements:

- $M(x,h)$ uses $M(y,h-1)$ but never $M(y,h-2)$. Thus we use $Mcur(x)$ to hold $M(x,h)$ and $Mlast(y)$ to hold $M(y,h-1)$ during iteration h.
- The starting nodes, S, just changes the estimate by summing over $x \in S$ instead of $x \in V$. In terms of implementation, this can be done by extending $Mcur$ to hold a *marked* bit indicating membership in S.

- The concluding nodes change the $h = 0$ case. Now $\mathcal{M}(x, 0)$ is $\{\}$ if $x \notin C$ since it can reach no nodes in C in zero steps. Thus nodes not in C are initially assigned a bitmask of all 0s.

The ANF-0 algorithm meets all but one of the requirements set out in the introduction:

Error guarantees: each $IN^+(x, h, C)$ is provably estimated with low error with high confidence.
Fast: running time is $O((n + m)d)$ which we expect to be fast since d is typically quite small (verified in section 4).
Low storage requirements: only additional memory for $Mcur$ and $Mlast$.
Adapts to the available memory? No! We will address this issue in the next section.
Easily parallelizable: Partition the nodes among the processors and then each processor may independently compute $Mcur$ for each x in its set. Synchronization is only needed after each iteration.
Sequential scans of the edge file: Yes.
Estimates $IN(x, h)$: Yes, with provable accuracy.

3.2 ANF Algorithm

The ANF-0 algorithm no longer accesses the edges in random order, but we now access $Mcur$ and $Mlast$ in an effectively random order. When we see the edge (x, y) we read and write $Mcur(x)$ and read $Mlast(y)$. If these tables are larger than the available memory, swapping will kill performance. We propose a small amount of preprocessing, to make these accesses predictable. Our idea is to break the large bitmasks $Mcur$ and $Mlast$ into b_1 and b_2, resp., equal-sized pieces. We partition the edges into $b_1 \times b_2$ buckets. In most cases, a one pass bucket sort can be used to partition the edges. Given that we have partitioned the edges, we would like to run the following algorithm to update $Mcur$:

FOR each bucket i of $Mcur$ DO
  Load bucket i of $Mcur$
  FOR each bucket j of $Mlast$ DO
    Load bucket j of $Mlast$
    FOR each edge (x, y) in bucket (i, j) DO
      $Mcur(x) = Mcur(x)$ OR $Mlast(y)$
  Write bucket i of $Mcur$

The cost of this algorithm is exactly the same cost as running ANF-0 plus the cost of the I/O (we have simply reordered the original computation). If $Mcur$ and $Mlast$ are N bytes long, then the cost of the I/O required to update the bitmasks is: $2N$ to load and store each bucket of $Mcur$ and $b_1 N$ to read $Mlast$ once for each bucket of $Mcur$. That is, the cost of this I/O is $(b_1 + 2)N$. Thus, we select b_1 and b_2 such that b_1 is minimal, given that we have enough file descriptors to efficiently perform the bucket sort in one pass.

Note that by reordering the computation to bucketize the edges, we now have very predictable I/O. Thus, we will insert *prefetching* operations which allows the computation and the I/O to be performed in parallel. The complete algorithm with prefetching appears in Figure 5.

This algorithm now meets all of our requirements.

Select the number of buckets $b1$ and $b2$
Partition the edges into the buckets (sorted by bucket)
FOR each node x DO
  IF $x \in C$ THEN
    $Mcur(x) =$ concatenation of k bitmasks each with 1 bit set ($P(\text{bit } i) = .5^{i+1}$)
  IF $x \in S$ THEN $mark(Mcur(x))$
  IF current buffer is full THEN
    switch buffers and perform I/O
Flush any buffers that need to be written
FOR each distance h DO
  Fetch the data for the first bucket of $Mcur$ and $Mlast$
  Prefetch next buckets of $Mcur$ and $Mlast$
  FOR each edge (x, y) DO
    IF $Mcur(y)$ is not in memory THEN
      We have been flushing and prefetching it
      Wait for it if necessary
      Asynchronously flush modified buffer
      Begin prefetching next buffer
    IF $Mlast(x)$ is not in memory THEN
      We have been prefetching it
      Wait for it if necessary
      Begin prefetching next buffer.
    $Mcur(x) = (Mcur(x)$ OR $Mlast(y))$
  // Copy $Mcur(u)$ to $Mlast(u)$ as we stream through $Mcur(u)$
  // computing the estimate
  est = 0
  Fetch the data for the first bucket of $Mcur$
  FOR each node x DO
    IF $Mcur(x)$ is not in memory THEN
      We have been prefetching it
      Wait for it to be available
      Start prefetching the next buffer
    $Mlast(x) = Mcur(x)$
    If x is the last element in its bucket of $Mlast$ THEN
      Asynchronously flush the buffer
      Continue processing in the double buffer
    IF $marked(Mcur(x))$ THEN
      $IN^+(x, h, C) = (2^b)/.77351$
      est += $IN^+(x, h, C)$
        where b is the average position of the least zero bits in the k bitmasks
  output $\hat{N}^+(h, S, C) = est$

Figure 5: ANF: Disk based processing

3.3 Leading Ones Compressions (ANF-C)

ANF is an algorithm that will be dominated by the I/O costs for large data sets and by the cost of the bit operations for smaller data sets. In both cases, we can further improve ANF by reducing the number of bits of data that must be manipulated. First, observe that, as ANF runs, most of the bitmasks will gradually accumulate a relatively lengthy set of leading 1s. That is, the bitmasks are of the form:

$$1111111110xxxxxx$$

It is wasteful to apply the bit operations and to write these leading 1s to disk. Instead, we will compress them. Second, we can achieve even better compression by *bit shuffling*. We have k parallel approximations, each of which has many leading ones. Instead of compressing each mask individually, we interleave the bitmasks by taking the first bit of each mask, followed by the second bit of each, etc. For example, with 2 masks:

$$11010, 11100 \Rightarrow 1111011000$$

which gives rise to a larger number of leading ones. The ANF-C algorithm uses a counter of the leading ones to reduce the amount of I/O and the number of bit operations. Like the *mark* bit, this counter can be prepended to the bitmask. In our experiments, we will find that leading ones compressions provide a significant speed-up, up to 23% in Figure 1.

4. EXPERIMENTAL EVALUATION

In this section we present an experimental validation of our ANF approximation. Two alternative approaches will be introduced and then we will describe our data sets. Next, we propose a metric for comparing two neighbourhood functions (functions over a potentially large domain). We conduct a sensitivity analysis of the parameter r. Then, we pick a value of r and we compare ANF to the approximation presented in [4] for various settings of the parameter k. We then show that sampling can provide very poor estimates and, finally, we examine the scalability of all approaches. The key results from this section are to answer these questions:

1. Is ANF sensitive to r, the number of extra bits?
2. Is ANF faster than existing approaches?
3. Is ANF more accurate than existing approaches?
4. Does ANF really scales to very large graphs? Do the others?

4.1 Framework

4.1.1 RI approximation scheme

The RI approximation algorithm is based on the approximate counting scheme proposed in [4]. To estimate the number of distinct elements in a multi-set, assign each a random value in $[0, 1]$ and record the least of these values added to the set. The estimated size is the reciprocal of the least value seen, minus 1. This approximate counting scheme was used to estimate the individual neighbourhood functions with the following algorithm. We need to know for each node, u the minimum value v_h of a node reachable from u in h hops. Then, the estimate for $IN(u, h)$ is $\frac{1}{v_h} - 1$. An equivalent, but more efficient algorithm was presented which uses breadth-first searches. It was shown that this improved procedure takes only $O(m \log n)$ time (with high probability). To reduce the variance in the estimates, the entire algorithm is repeated, averaging over the estimates.

4.1.2 Sampling

We can sample by selecting random edges, random nodes or random starting nodes for the breadth-first search. Randomly selecting a set of nodes (and all edges for which both end-points are in this set) and randomly selecting a set of edges is unlikely to produce a useful sample. For example, imagine sampling a cycle – anything but a very large sample will leave disconnected arcs which have very different properties. For completeness we verified that these approaches produced useless estimates. The last approach is much more compelling. It is akin to the sampling done in [14]. Recall that the neighbourhood function is: $N(h) = \sum_{u \in V} IN(u, h)$. Rather than summing over all nodes, u, we could sum over only a sample of the nodes while using breadth-first searches to compute the exact $IN(u, h)$. We call this method *exact-on-sample* and it has the potential to provide great estimates – a single sample of a cycle will provide an exact solution. However, experimentally we find that this approach also has the potential to provide very poor estimates. Additionally, we find that it does not scale to large graphs because of the random-like access to the edge file due to its use of breadth-first search.

Table 2: Data set characteristics

Name	(n) #Nodes	(m) #Edges	Degree Max.	Degree Avg.	Prac. Diam.	Orient.
Cornell	844	1,647	131	1.95	9	Dir
Cycle	1,000	1,000	2	2.00	500	Undir
Grid	10,000	19,800	4	1.98	100	Undir
Uniform	65,378	199,996	20	3.06	8	Undir
Cora	127,083	330,198	457	2.60	35	Dir
80-20	166,946	449,832	723	2.69	10	Undir
Router	284,805	430,342	1,978	1.51	13	Undir

4.1.3 Experimental Data Sets

We have collected three real data sets and generated three synthetic data sets. These data sets have a variety of properties and cover many of the potential applications of the neighbourhood function. Some summary information is provided in Table 2. "Prac. Diam." is the *Practical Diameter* which we use informally to mean the distance which includes most of the pairs of points. We use three real data sets:

Router: Undirected Internet routers data from ISI [19], including scans done by Lucent Bell Laboratories [12].
Cornell: A crawl of the Cornell web site by Mark Craven.
Cora: The CORA project at JustResearch found research papers on the web and provided a citation graph [6].

and four synthetic data sets:

Cycle: A single simple undirected cycle (circle).
Grid: A 2D planar grid (undirected).
Uniform: Graph with random (undirected) edges.
80-20: Very skewed graph generated in an Internet like fashion with undirected edges using the method in [17].

4.1.4 Evaluation Metric

We are approximating functions defined over d points. Let N be the true neighbourhood function and $\hat{N}$ be the candidate approximation. To measure the error of $\hat{N}(h)$, we use the standard relative error metric. To measure the overall error of $\hat{N}$ we use the Root Mean Square (RMS) of pointwise relative errors. Thus, the error function, e, is:

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

Note that the RMS is computed beginning with $h = 2$. Since $N(0) = |V|$ and $N(1) = |E|$ we do not require approximations for these points.

4.2 Results

4.2.1 Parameter Sensitivity

ANF has two parameters: the number of parallel approximations, k, and the number of additional bits, r. The number of approximations, k, is a typical trade-off between time and accuracy. We consider this in section 4.2.2 and fix $k = 64$ for the time being. Additional experiments were run with other values of k which produced similar results. To measure the sensitivity we averaged the RMS error over 10 trials for different values of r and the different data sets. These results appear in Figure 6 and we see that the accuracy is not very sensitive to the value of r. (The lines between the

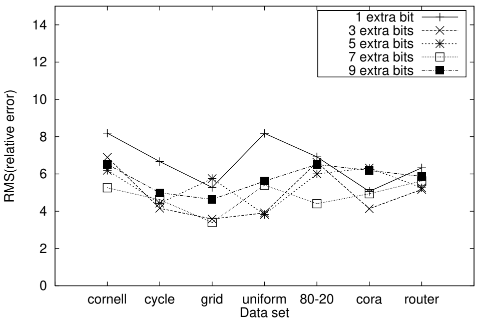


Figure 6: Results are not sensitive to r

points are a visual aid only.) We find $r = 5$ or $r = 7$ provide consistent results.

4.2.2 *Accuracy*

We now examine the accuracy of the ANF approximation. To do so, we compare our accuracy with a highly tuned implementation of the RI approximation (the only existing approach). Now we fix $r = 7$ and consider three values of k: 32, 64 and 128. We average the error over 10 trials of each approximation scheme. The first row of Figure 7 shows the accuracy of each of the k values for each data set for each algorithm, while the second row shows the corresponding running times. We see that:

- ANF's error is independent of the data sets.
- RI approximation's error varies significantly between data sets.
- ANF achieves less than 10%, 7% and 5% errors for $k = 32$, $k = 64$ and $k = 128$, respectively.
- RI has errors of 27%, 14% and 12% for $k = 32$, $k = 64$ and $k = 128$, respectively.
- ANF is much faster than RI, particularly on the larger graphs, with up to 3 times savings.
- Using much less time, ANF is much more accurate than RI.

Thus, even for the case of graphs that may be stored in memory, we have a significant improvement.

4.2.3 *Sampling*

There are three problems with the described *exact-on-sample* approach. First, it has *heavy memory requirements* because fast breadth-first search requires that the edge file fit in memory. Second, the *quality is dependent on the graph* because there are no bounds on the error. Third, it is not possible to compute the *individual neighbourhood functions*. We now provide an example which demonstrates the first two problems. Figure 8(a) helps illustrate our example graph. 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 graph 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 $N \gg s$ and $N \gg t$, then the majority of the sampled nodes will be from the N center nodes and very few will be from the s source nodes. This will result in an error that is a factor of around s/p for exact-on-sample using a p% sample. We measure the error and the running time over 20 trials for a variety of sample sizes ranging from .1% to 15% on a graph generated with $N = 25,000$, $s = 100$, $t = 100$ and $d = 6$. Figure 8(b) shows the large errors, more than 20%, even for very large samples.

To illustrate the scalability issues for exact-on-sample, we constructed a graph with $N = 250,000$, $s = t = 5$ and $d = 6$. We then increase s and t proportionately to generate larger graphs. Figure 8(c) shows that as the graph grows larger exact-on-sample scales about as well as ANF but as soon as the edge file no longer fits in memory (approximately 27 million edges) we see approximately a two order of magnitude increase in the running time of the exact-on-sample approach. Thus, we conclude that exact-on-sample scales very poorly to graphs that are larger than the available memory.

Table 3: Wall clock running time (minutes)

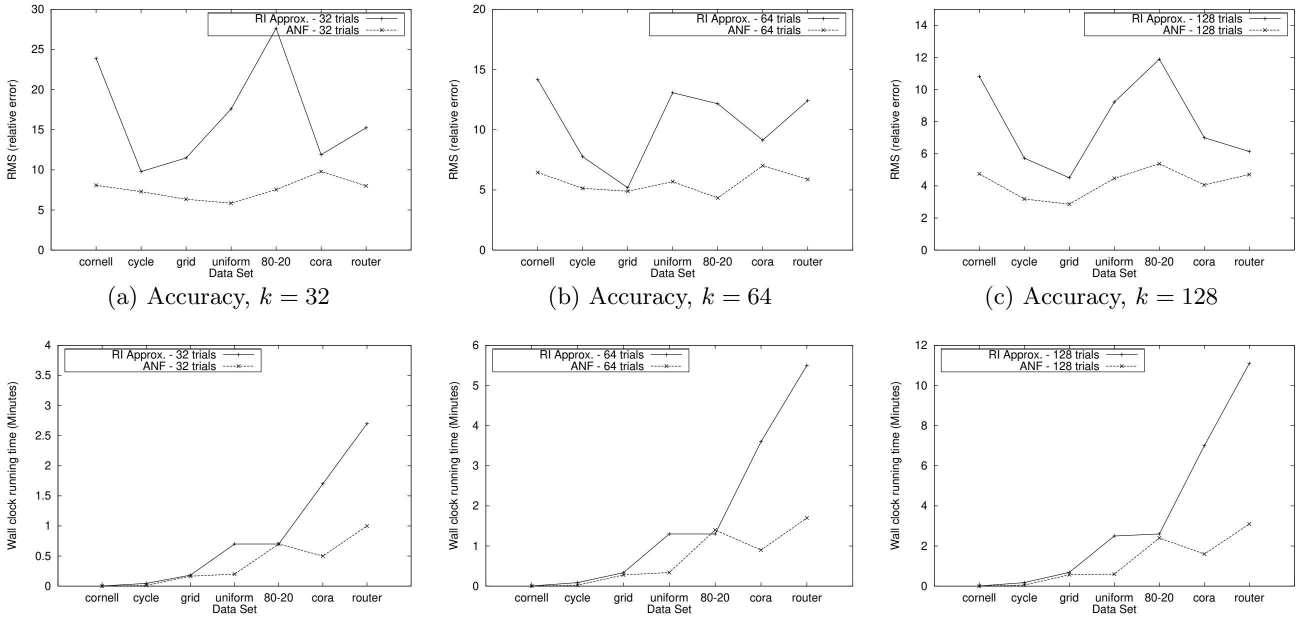
Data Set	BF (Exact)	ANF	Speed-up
Uniform	92	0.34	270x
Cora	6	1.4	4x
80-20	680	0.9	756x
Router	1,200	1.7	705x

4.2.4 *Speed and Scalability*

Table 3 reports wall-clock running times on an otherwise unloaded Pentium II-450 machine for both the exact computation (Breadth-First search) and ANF with $k = 64$ parallel approximations. We chose $k = 64$ since it provides much less than a 10% error, which should be acceptable for most situations. The approximations are quite fast and, for the *Router* data set, we have reduced the running time from approximately a day down to less than 2 minutes. This makes it possible to run drill down tasks on much larger data sets than before. Overall, we find that ANF is up to 700 times faster than the exact computation on our data sets.

ANF also scales to much larger graphs than the alternatives. We generated random graphs placing edges randomly between nodes. We increased the number of nodes and edges while preserving an edge:node ratio of 8:1 (based on the average degree found in two large crawls of the Web [3]). Figure 1 (in the introduction) shows the running times for the ANF variants, the RI approximation and example-on-sample. Parameters for each alternative were chosen such that they all had approximately the same running time for the first data point. These values are $k = 32$ for the ANF variants, $k = 8$ for RI and $p = 0.0015$ for exact-on-sample. We find that:

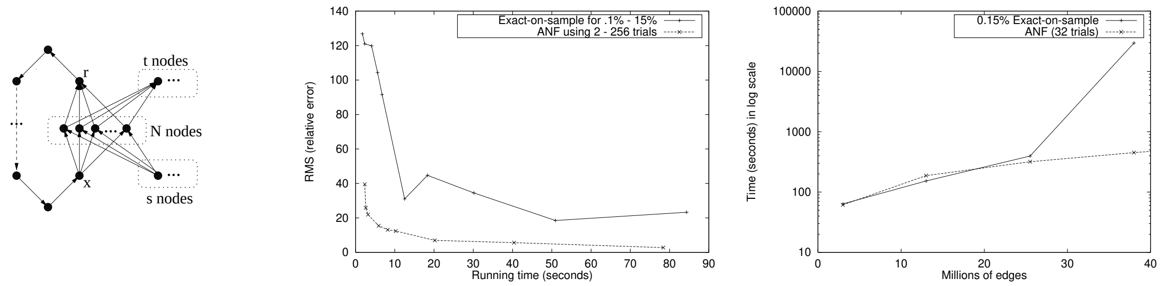
1. Exact-on-sample scales much worse than linearly. For a fixed sampling rate, we expect it to scale quadratically when we increase the number of nodes and edges.
2. RI very quickly exhausts its resources due to its data structures. Because RI was not designed to avoid the random accesses, it has horrible paging behaviour and, after about 2 million edges, we had to stop its timing experiment.
3. ANF-0 suffers from similar swapping issues when it



(a) Accuracy, $k = 32$ (b) Accuracy, $k = 64$ (c) Accuracy, $k = 128$

(d) Time, $k = 32$ (e) Time, $k = 64$ Time (f), $k = 128$

Figure 7: Our ANF algorithm provides more accurate and faster results than the RI approximation



(a) Graph exhibiting huge errors (b) Error vs. Running time (c) Running time vs. # of edges

Figure 8: Sampled breadth-first search can provide huge errors and does not scale to very large edge files

exhausts the memory at around 8 million edges, and it too had to be stopped.

4. Approximate counting methods [9, 4] are not enough for disk resident graphs.
5. ANF/ANF-C scale the best, growing piece-wise linearly with the size of the graph. The break points are: all data fits in memory (about 8 million edges), $Mcur$ fits in memory (about 16 million edges) and neither fits in memory (the rest). This is as expected.
6. ANF-C offers up to a 23% speed-up over ANF.

Thus, ANF is the only algorithm that scales to large graphs and does so with a linear increase in running time.

5. DATA MINING WITH OUR ANF TOOL

With our highly-accurate and efficient approximation tool, ANF, it is now possible to answer some of the prototypical graph mining questions that we posed in the introduction. Due to the limits of page constraints and data availability, we will report on answers to only a representative sample of those questions. However, all 10 questions can be answered by the same approaches that we will now demonstrate. The approach is to compute various neighbourhood functions and then to compare them. Our tool allows for a detailed comparison of these functions. However, comparing neighbourhood functions requires that we compare two functions over potentially large domains (the domain is $\{1, 2, \cdots, d\}$). Instead, in this paper we will focus on a summarized statistic derived from the neighbourhood function, called the *hop exponent*. Many real graphs [8] have a neighbourhood function that follows a power law $N(h) \propto h^{\mathcal{H}}$. The exponent, $\mathcal{H}$, has been defined as the *hop exponent* (similarly, $\mathcal{H}_x$ is the *individual hop exponent* for a node x).

There are three interesting observations about the hop exponent that make it an appealing metric. First, if the power-law holds, the neighbourhood function will have a linear section with slope $\mathcal{H}$ when viewed in log-log scale. Second, the hop exponent is, informally, the *intrinsic* dimensionality

2.50	2.37	2.50
2.36	**2.57**	2.36
2.50	2.36	2.51

(a) ANF importance

-.07	-.20	-.07
-.21		-.21
-.07	-.21	-.06

(b) Delta importance

Figure 9: ANF finds the best starting move for `X`

of the graph. A cycle has a hop exponent of 1 while a grid has a hop exponent of 2, which corresponds with some idea of their dimensionality. Third, if two graphs have different hop exponents, there is no way that they could be similar. While not all neighbourhood functions will actually follow a power-law, we have found that using the hop exponent still fairly reasonably captures the growth of the neighbourhood function.

To compute the hop exponent, we first truncate the neighbourhood function at d_{eff}, the *effective diameter*, then we compute the least-squares line of best fit in log-log space to extract the slope of this line. The slope is the hop exponent of the graph and we use it as our measure of the growth of a neighbourhood function. We define d_{eff} to be the least h such that it include 90% of the pairs of nodes. We use the individual hop exponent, $\mathcal{H}_x$, as a measure of a node's importance with respect to the connectivity of a graph. We can answer some of the proposed questions.

5.1 Tic-Tac-Toe

Tic-tac-toe is a simple game in which two players, one using `X` and the other using `O`, alternatively place their mark on an unoccupied square in a 3x3 grid. The winner is the first player to connect 3 of their symbols in a row, column or diagonal. The best opening move for `X` is the center square, the next best is any of the 4 corners and the worst moves are the 4 remaining squares. To verify that our notion of importance has some potential use, we will use our ANF tool to discover this same rule. Construct a graph where each node is a valid board and add an edge from board x to board y to indicate that this is a possible move. Let C, the concluding set, be the set of all boards in which `X` wins. Compute the individual neighbourhood functions for each of the 9 possible first moves by `X`, which is their importance (speed at which they attain winning positions from each of these moves). Figure 9 shows these importances along with the difference between each and the best move. ANF determined the correct importance of each opening move. Using Figure 9(b), we see that the center is only slightly better than a corner square which is, in turn, much better than the remaining 4 squares. This shows both the correct ordering of the starting moves and the relative importance of each.

5.2 Clustering movie genres

The Internet Movie Data Base (IMDB) [10] is a collection of relations about movies. We will map a subset of the relations into graphs to illustrate questions that we can now answer thanks to our ANF tool. First, construct the `actor-movie` graph by creating a node for each actor and a node for each movie. An undirected edge is placed between an actor, a, and a movie, m, to indicate an appearance by a in m. Let S be the nodes we created for the actors. We now employ another relation of the IMDB. Each movie has been identified as being in one or more genres (such as documentaries,

Film-Noir
Animation Short
Adult
Fantasy Documentary Family
Mystery Musical Western War
Sci-Fi
Romance
Horror Adventure Crime Thriller Action
Comedy
Drama

Figure 10: Movie genre clusters sorted in increasing hop exponent value

dramas, comedies, etc). For each genre, we take the set of movies in that genre and the set of actors which appear in one or more of those movies. We then cluster these graphs by computing the hop exponents and forming clusters that have similar hop exponents (less than 0.1 difference). This clustering appears in Figure 10. One interesting cluster is `mystery`, `musical`, `western`, `war` which actually corresponds to movies that are typically older. Finally, other fringe genres such as `Adult` turn out to be well separated from the others.

5.3 Internet Router Data

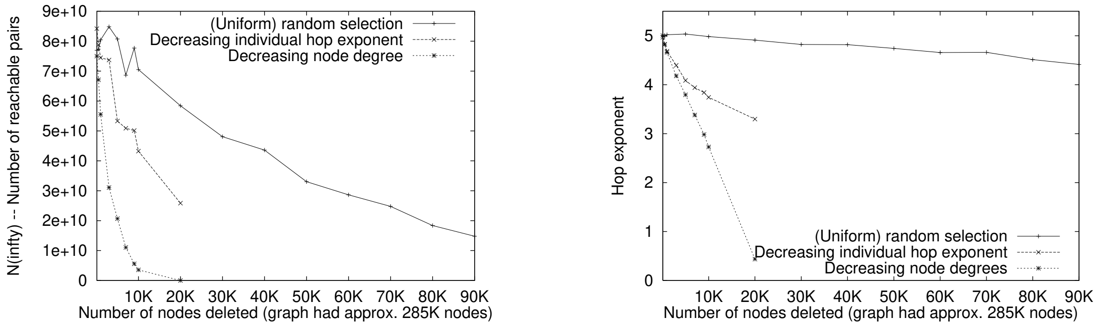
In the networking community, a study used an early version of our ANF tool (ANF-0) to look at the inherent robustness of the Internet. That is, the robustness that we observe from the topology itself. Each router was a node and edges were used to indicate communication links. Here we reproduce some of the results of this study, including Figure 11, from [16]. The goal is to determine how robust the Internet is to router failures. As an experiment, we delete some number of routers and then measure the total connectivity (number of pairs of routers that are still able to communicate) and the hop exponent of the graph. The three lines differ in how the deleted routers are selected. First, randomly selected nodes are deleted. Second, nodes are deleted in decreasing order of their importance. Third, routers are deleted in decreasing order of their degree. Here we see some very interesting results:

1. Random failures do not disrupt the Internet.
2. It may be possible to take a random sample of the Internet by deleting random routers and adjacent edges. This appears possible because we found that the connectivity information (hop exponent) is not significantly changed under random deletions.
3. Targeted failures of routers can very quickly and very dramatically disrupt the Internet.

This type of study was infeasible before our ANF tool as an exact computation would have required over **a year of computation time**.

6. CONCLUSIONS

In this paper we presented 10 interesting data mining questions on graph data, proposed an efficient and accurate approximation algorithm that gives us the tool, ANF, we needed to answer these questions, and presented results for three of these questions on real-world data. We have found ANF to be quite useful for these and other questions that can be addressed by studying the neighbourhood structure of the



(a) Number of pairs of nodes that can communicate vs. number of deleted nodes

(b) Hop exponent of the Internet vs. number of deleted nodes

Figure 11: Effect of router failures on the Internet

underlying graphs (e.g., we have used ANF to study the most important movie actors). We experimentally verified that ANF provides the following advantages:

Highly-accurate estimates: Provable bounds which we also verified experimentally, finding less than a 7% error when using $k = 64$ parallel approximations (for all our synthetic and real-world data sets).

Is orders of magnitude faster: On the seven data sets used in this paper, our algorithm is up to 700 times faster than the exact computation. It is also up to 3 times faster than the RI 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 a disk-based version of our algorithm and experimentally verified that it scales with the graph size.

Can be parallelized: Our ANF algorithm may be parallelized with very few synchronization points.

Employs sequential scans: Unlike prior approximations of the neighbourhood function, our algorithm avoids random access of the edge file and performs one sequential scan of the edge file per hop.

Individual neighbourhood functions for free: ANF computed approximations of the individual neighbourhood functions as a byproduct of the computation. These approximations proved to be very useful in identifying the "important" nodes in a graph.

Even for the case that graphs (and data structures) fit into memory, ANF represents a significant improvement in speed and accuracy. When graphs get too large to be processed effectively in main memory, ANF makes it possible to answer questions that would have been at least infeasible, if not impossible, to answer before. In addition to its speed, we found the neighbourhood measures to be useful for discovering the following answers to our prototypical questions:

1. We found the best opening moves to tic-tac-toe.
2. We clustered movie genres.
3. We found that the Internet is resilient to random failures while targeted failures can quickly create disconnected components.
4. We found that sampling the Internet actually preserves some connectivity patterns while targeted failures truly distort it.

7. REFERENCES

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

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[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] Cook and Holder. Graph-based data mining. *ISTA: Intelligent Systems & their applications*, 15, 2000.

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

[7] P. Domingos and M. Richardson. Mining the network value of customers. In *KDD-2001*, pages 57–66, 2001.

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

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

[10] IMDB. `http://www.imdb.com`.

[11] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PDKK*, pages 13–23, 2000.

[12] `http://cs.bell-labs.com/who/ches/map/`.

[13] 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.

[14] 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.

[15] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proc. ACM PODS Conference (PODS-93)*, pages 222–232, 1993.

[16] C. R. Palmer, G. Siganos, M. Faloutsos, C. Faloutsos, and P. Gibbons. The connectivity and fault-tolerance of the Internet topology. In *Workshop on Network-Related Data Management (NRDM-2001)*, 2001.

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

[18] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

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

[20] S. L. Tauro, C. Palmer, G. Siganos, and M. Faloutsos. A simple conceptual model for the Internet topology. In *IEEE Globecom 2001*, 2001.