Alcom-FT Technical Report Series
ALCOMFT-TR-03-120

Algorithms and Experiments for the Webgraph*

Luigi Laura † Stefano Leonardi * Stefano Millozzi * Ulrich Meyer‡
Jop F. Sibeyn §

June 27, 2003

Abstract

In this paper we present an experimental study of the properties of web graphs. We study a large crawl from 2001 of 200M pages and about 1.4 billion edges made available by the WebBase project at Stanford [26], and synthetic graphs obtained by the large scale simulation of stochastic graph models for the webgraph.

This work has required the development and the use of external and semi-external algorithms for computing properties of massive graphs, and for the large scale simulation of stochastic graph models. We report our experimental findings on the topological properties of such graphs, describe the algorithmic tools developed within this project and report the experiments on their time performance.

*Partially supported by the Future and Emerging Technologies programme of the EU under contracts number IST-2001-33555 COSIN "Co-evolution and Self-organization in Dynamical Networks" and IST-1999-14186 ALCOM-FT "Algorithms and Complexity in Future Technologies", and by the Italian research project ALINWEB: "Algorithmica per Internet e per il Web", MIUR – Programmi di Ricerca di Rilevante Interesse Nazionale.

†Dip. di Informatica e Sistemistica Universita' di Roma "La Sapienza" Via Salaria 113 00198 Roma Italy. E-mail: {laura,leon,millozzi}@dis.uniroma1.it

‡Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. E-mail: umeyer@mpi-sb.mpg.de

§Halle University, Institute of Computer Science, Von-Seckendorff-Platz 1, 06120 Halle Germany. E-mail:jopsi@informatik.uni-halle.de

1 Introduction

Networks are widely used to represent relationships or physical connections in the Internet. In this work we concentrate our attention on the *Webgraph*, the graph whose nodes are (static)web pages and edges are (directed) hyperlinks among them. The *Webgraph* has been the subjet of a large interest in the scientific community. The reason of such large interest is primarily given to search engine technologies. Data mining on the web has greatly benefited from the analysis of the link structure of the Webgraph. One example is represented by the algorithms for ranking pages such as PageRank [5] and HITS [13]. Link analysis is also at the basis of the sociology of content creation, and the detection of cyber-communities hidden in the Web [16].

A large amount of research has recently been focussed on studying the properties of the Webgraph by collecting and measuring samples spanning a good share of the whole Web. A second important research line has been the development of stochastic models generating graphs that hold the properties of the Web. Developing a realistic and accurate stochastic graph model able to reproduce several topological properties of the Webgraph is relevant for testing web applications, detecting peculiar regions of the Webgraph, and predicting the evolution of new phenomena in the Web.

This research work also poses several algorithmic challenges. It requires to develop algorithmic tools to compute topological properties on graphs of several billion edges [1].

1.1 Properties and models of the Webgraph

The Webgraph has shown the ubiquitous presence of power law distributions, a typical signature of scale-free properties. Barabasi and Albert [3]and Kumar et al [16] suggested that the distribution of the indegree and of the outdegree of the Webgraph follow a *power-law* distribution (see Figure 7.). Later experiments by Broder et al. [6] on a crawl of 200M pages from 1999 by Altavista confirmed it as a basic property. The probability that the indegree of a vertex is i is distributed by a power-law, Pr_u[in-degree(u)= i]$\propto 1/i^{\gamma}$, for $\gamma \approx 2.1$. The outdegree of a vertex was also observed [6]to be distributed according to a power law with exponent roughly equal to 2.7 (see Figure 7). The number of edges observed in the Webgraph is about equal to seven times the number of vertices.

Broder et. al. [6] also presented a fascinating picture of the Web's macroscopic structure: a *bow-tie* shape (see Figure 8), where the major part of the sites can be divided into three sets: a core made by a large strongly connected component (SCC) of about 28% of the vertices, i.e., sites that are mutually connected to each other, and two sets (IN and OUT), each one containing about 22 % of the vertices, made by the sites that can only reach (or be reached by) the sites in the SCC set. They also estimated the probability of the existence of a path between a random source and a random destination. They found out that the Webgraph exhibits the *small world phenomenon* [25, 12] only if the hyperlinks are considered undirected: almost all pages of a giant connected component including about 90% of the web documents are reachable within few hops from every other page.

A surprising number of specific topological structures such as bipartite cliques of relatively small size, from 3 to 10, has also been recognized in the Webgraph [16]. The study of such structures is aimed to trace the emergence of hidden *cyber-communities*: groups of individuals pointing to pages that are relevant for their common interests. A bipartite clique is interpreted as a core of such a community, defined by a set of fans, all pointing to a set of centers/authorities for a given subject, and the set of centers, all pointed to by the fans. Over 100,000 such communities have been recognized [16] on a sample of 200M pages on a Web crawl from Alexa of 1997 as shown in Figure 9.

The Google search engine is based on the popular PageRank algorithm first introduced by Brin

[1]A study made by Cyvellance [8] showed that in July 2000 the Web reached 2.1 billion webpages (indexable), and this number is growing by 7 million pages each day.

and Page [5]. The PageRank distribution has a simple interpretation in terms of a random walk in the Webgraph. Assume the walk has reached page p. The walk then continues either by following with probability $1-c$ a random link in the current page, or by jumping with probability c to a random page. Pagerank can be seen as the stationary probability of such process, or equivalently, for every vertex, the probability that the random walk stops at that vertex. The correlation between the distribution of PageRank and in-degree has been recently studies in a work of Pandurangan, Raghavan and Upfal [20]. They show by analyzing a sample of 100,000 pages of the brown.edu domain that PageRank is distributed with a power law of exponent 2.1 . This exactly matches the in-degree distribution, but very surprisingly they also show very little correlation between these quantities ,i.e., pages with high in-degree may have very low PageRank.

The topological properties observed in the WebGraph cannot be found in the traditional random graph model of Erdös and Rényi (ER) [9]. Firstly, the ER model is a static model, while the Webgraph evolves over time when new pages are published or are removed from the Web. Secondly, the ER model fails to capture basic properties since for instance the distribution of the in-degree is sharply concentrated around its mean.

Aiello, Chung and Lu [1] proposed a static stochastic graph model appropriately customized to reproduce the power law distribution on the degree of the vertices.

Albert, Barabasi and Jeong [2] initiated the study of evolving networks by presenting a model in which at every discrete time step a new vertex is inserted in the graph. The new vertex connects to previously inserted vertices with a constant number of edges; a vertex is chosen as endpoint of an edge with the so called *preferential attachment* rule, i.e. with probability proportional to its in-degree. This model shows a power law distribution over the in-degree of the vertices with exponent roughly 2 when the number of edges that connect every vertex to the graph is 7. In the following sections we refer to this model as the *Evolving Network (EN)* model.

The Copying model has been later proposed by Kumar et al. [15] to explain other relevant properties observed in the Webgraph such as a large number of bipartite cliques of small size. For every new vertex entering the graph it is selected at random a prototype vertex p. A constant number d of links connect the new vertex to previously inserted vertices. The model is parameterized on a *copying factor* α. The end-point of the lth link, $l = 1, ..., d$, is either copied with probability α from the corresponding lth out-link of the prototype vertex p or it is selected at random with probability $1 - \alpha$. The copying event aims to model the formation of a large number of bipartite cliques in the Webgraph. In our experimental study we consider the *linear*[15] version of this model, and we refer to it simply as the *Copying* model.

Very recently Pandurangan, Raghavan and Upfal [20] proposed a model that complements the Evolving Network model [2] by choosing the endpoint of a link with probability proportional to the in-degree and to the PageRank of a vertex. The authors show by computer simulation that with an appropriate fitting of the parameters the graphs generated capture the distributional properties of both PageRank and in-degree.

More models of the Webgraph are presented by Pennock et al. [21], Caldarelli et al. [17], Cooper and Frieze [7]. Mitzenmacher [19] presents an excellent survey of generative models of for power-law distributions. Bollobás and Riordan [4] study vulnerability and robustness of scale-free random graphs. Most of the models presented in the literature generate graphs without cycles. Albert et al. [2] amongst others proposed to rewire part of the edges introduced in previous steps to insert links to later inserted vertices.

2 Our Results

In this work we present an extensive study of the statistical properties of the Webgraph by analyzing a crawl of about 200M pages collected in 2001 by the Web Base project at Stanford [26] and made available for our study. We also report new results on stochastic graph models for the Webgraph presented in literature.

This work has required the development of semi-external memory algorithms for computing disjoint bipartite cliques of small size, external memory algorithms based on the ideas of [10] for computing PageRank, and the large scale simulation of stochastic graph models. Moreover, we use the semi-external algorithm developed in [22] for for computing Depth First Search and Strongly Connected Components.

Our principal findings are:

1. Accordingly to previous studies, we observed the in-degree distributed with a power law of exponent 2.1. Differently from previous studies, we noted that the outdegree is not distributed with a power law.

2. We confirm on much larger scale the observation of Panduragan [20] on a sample of 100,000 from the brown.edu that PageRank is also distributed with a power law with exponent 2.1. More importantly, we also confirm the absence of correlation between PageRank and in-degree.

3. We have observed in the Web Base crawl an increased number of disjoint bipartite cliques with respect to the study [16] on the Alexa crawl of 200M pages from 1997. The number is higher by one order of magnitude. This follows either by the increase of the number cyber communities, or by the better performance of our semi-external heuristic for computing bipartite cliques.

4. We rewired edges of the graphs generated according to the Evolving Networks [2] and to the Copying model [15] in order to induce cycles. The rewiring process consists of picking a random edge of the initial graph and replace it with an edge whose two endpoints are selected uniformly at random. We observed that the size of the largest SCC to *smoothly* increases, and the number of SCCs to smoothly decreases, with the number of the edges of the graph that are rewired. The largest SCC spans a big part of the graph when the number of rewired edges is about three times the number of vertices of the graph. Imposing a random graph over a stochastic graph model for the Web then results in a behavior that is completely different from the *threshold phenomena* observed in the traditional Erdös Renyi model. This can be explained with the presence in the graph of a number of vertices of high degree that form the skeleton of the strongly connected component.

5. We empirically study the time performance of the external memory algorithms for computing PageRank, SCCs and disjoint bipartite cliques. We run these algorithms on graphs generated according to the evolving network [2] and the copying linear model [15] of size up to 60M vertices and different percentage of rewired edges. The time performance of all the algorithms grows linearly with the size of the graph.

Software tools developed within this project are presented in [18].

The analysis of the Web Base crawl of the Webgraph is presented in Section 3. The analysis of the strongly connected components of stochastic graph models is reported in Section 4. Section 5 presents the external and semi-external algorithms.

3 Analysis of the Web Base crawl

We conducted our experiments on a $200M$ nodes crawl collected from the WebBase project at Stanford [26] in 2001. We have performed the following measures:

a The in-degree distribution of the graph.

b The out-degree distribution of the graph.

c The PageRank [5] distribution of the vertices of the graph.

d The correlation between the in-degree and the PageRank distribution.

e The number of disjoint bipartite cliques in the graph.

The in-degree distribution, shown in Figure 10, follows a power law with $\gamma = 2.1$. This confirms the observations done on the crawl of 1997 from Alexa [16], the crawl of 1999 from Altavista [6] and the notredame.edu domain [3].

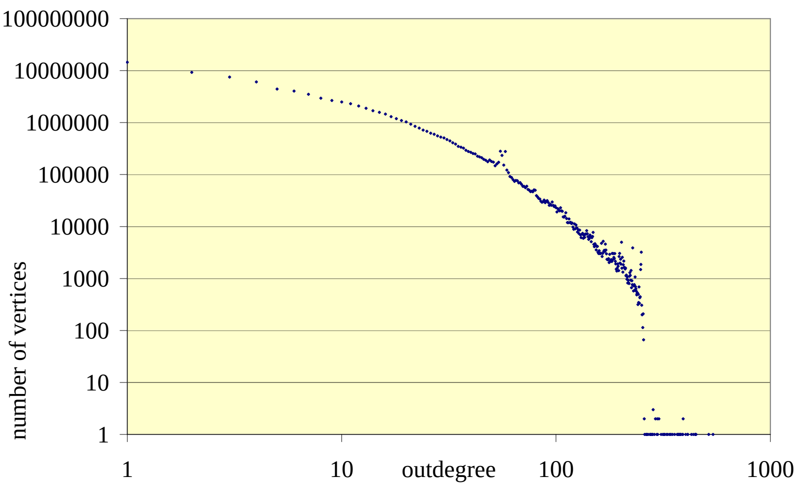

Figure 1: Out-degree distribution of the Web Base crawl

We note a *bump* between the values 1.000 and 10,000, that has also been observed by Broder *et al.* [6] and it is probably due to a huge clique created by a single *spammer*.

In Figure 1 the out-degree distribution of the WebBase crawl is shown. While the in-degree distribution is fitted with a power law, the out-degree is not. This does not agree for instance with the report of the observations on the Altavista crawl [6]. A possible explanation of this phenomena is that writing a scale-free series of hyperlinks is seriously limited by the patience of webmasters.

We present in Figure 11 a plot of the PageRank distribution of the Web Base crawl. Here, we confirm the observation of [20] on the scale free property of such quantity that is distributed according to a power-law with exponent $\gamma = 2.109$. We also computed the statistical correlation between PageRank and in-degree. We obtained a value of $-5.1877E-6$, on a range of variation in $[-1, 1]$ from negative to positive correlation. This confirms on much larger scale the observation done by [20] on the brown.edu domain of 100,000 pages that the correlation between the two measures is very weak.

In Figure 2 the graphic of distribution of the number of bipartite cliques (i, j), with $i, j = 1, \ldots, 10$ is shown. The shape of the graphic follows that one presented by Kumar et al. [16] for the 200M crawl by Alexa (see Figure 9), but with a difference in size: we detect a number of bipartite cliques of size $(4, j)$ that differs from the crawl from Alexa for more than one order of magnitude. A possible (and quite natural) explanation is that the number of *cyber-communities* has consistently increased from 1997 to 2001 (we recall that the longevity of *cyber-communities*' website is bigger as compared to other websites [16]). A second explanation is that our algorithm for finding disjoint bipartite cliques, which is explained in 5.1, is more efficient than the one implemented in [16].

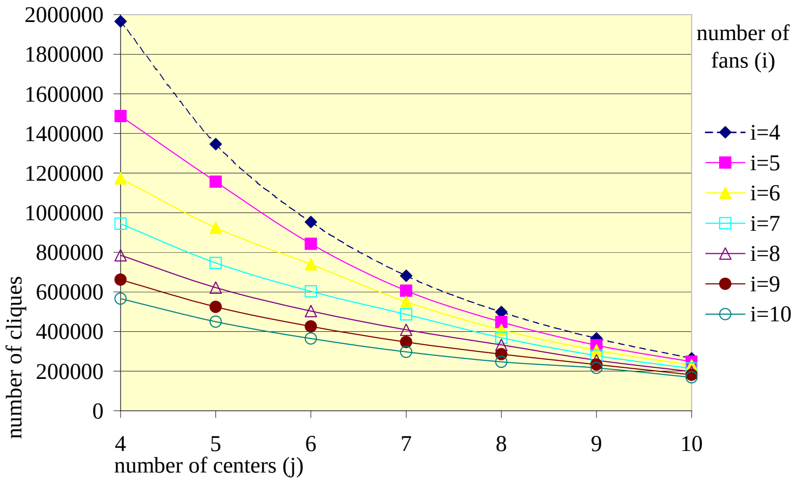

Figure 2: The number of bipartite cliques (i, j) in the Web Base crawl

4 Strongly connected components

Broder et al. [6] identified a very large strongly connected component including a large share, about 28%, of the entire crawl.

The Evolving Network and the Copying model do not contain cycles and hence not even a single strongly connected component. We therefore modified the EN and the Copying model by rewiring a share of the edges. The process consists of picking one ramdom edge of the graph and replace it with an edge whose end-points are chosen at random. The experiment consisted in rewiring a number of edges ranging from 1% to 300% of the number of vertices in the graph. Recall that these graphs contain 7 times as many edges as the vertices.

The most remarkable observation is that, differently from the Erdös-Renyi model, we do not observe any threshold phenomenon in the emerging of a large SCC. In a classical random graph, it is observed the emerging of a giant connected component when the number of edges grows over a threshold that is slightly more than linear in the number of vertices. The size of the largest SCC increases smoothly with the number of edges that are rewired up to spanning a big part of the graph. We also observe the number of SCCs to decrease smoothly with the increase of the percentage of rewired edges. This can be observed in Figures 3 and 12 for a graph of 10M vertices.

Devising strongly connected components in a graph stored on secondary memory is a non-trivial task for which we used a semi-external algorithm developed in [22]. This algorithm together with its time performance is described in Section 5.

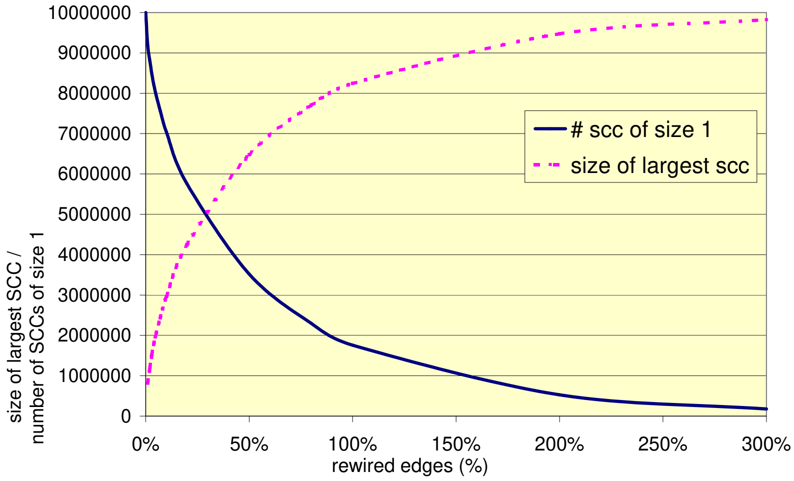

Figure 3: Number and size of SCCs – (Copying Model)

5 Algorithms for analyzing and generating Web graphs

In this section we present the external and semi-external memory algorithms we developed and used in this project for analyzing massive Webgraphs and their time performance. Moreover, we will present some of the algorithmic issues related to the large scale simulation of stochastic graph models.

For measuring the time performance of the algorithms we have generated graphs according to the Copying and the Evolving Network model. In particular, we have generated graphs of size ranging from 100,000 to 50M vertices with average degree 7, and rewired a number of edges equal to 50% and 200% of the vertices. The presence of cycles is fundamental for both computing SCCs and PageRank. This range of variation is sufficient to assess the asymptotic behaviour of the time performance of the algorithms. In our time analysis we computed disjoint bipartite cliques of size $(4, 4)$, the size for which the computational task is more difficult.

The analysis of the time complexity of the algorithms has been performed by restricting the main memory to 256 MB for computing disjoint bipartite cliques and PageRank. For computing strongly connected components, we have used 1GB of main memory to store a graph of 50M vertices with 12.375 bytes per vertex. Figures 4,5 and 6 show the respective plots. The efficiency of these external memory algorithms is shown by the linear growth of the time performance whenever the graph does not fit in main memory. More details about the data structures used in the implementation of the algorithms are given later in the section.

5.1 Disjoint bipartite cliques

In [16] an algorithm for enumerating disjoint bipartite cliques (i, j) of size at most 10 has been presented, with i being the fan vertices on the left side and j being the center vertices on the right side. (In [11] a semi-external algorithm for the maximum clique problem has also been presented.) The algorithm is composed of a pruning phase that consistently reduces the size of the graph in order to store it in main memory. A second phase enumerates all bipartite cliques of the graph. A final phase selects a set of bipartite cliques that form the solution. Every time a new clique is selected, all intersecting cliques are discarded. Two cliques are intersecting if they have a common fan or a common center. A vertex can then appear as a fan in a first clique and as a center in a second clique.

In the following, we describe our semi-external heuristic algorithm for computing disjoint bipartite cliques. The algorithm searches bipartite cliques of a specific size (i, j).

Two n-bit arrays Fan and $Center$ stored in main memory indicate with $Fan(v) = 1$ and $Center(v) = 1$ whether fan v or center v has been removed from the graph. We denote by $I(v)$ and $O(v)$ the list of predecessors and successors of vertex v. Furthermore, let $\tilde{I}(v)$ be the set of predecessors of vertex v with $Fan(\cdot) = 0$. We denote by $\tilde{O}(v)$ the set of successors of vertex v with $Center(\cdot) = 0$. Finally, let $T[i]$ be the first i vertices of an ordered set T.

In the following we first give an outline of the idea underlying the algorithm. Consider a fan vertex v with at least j successors with $Center(\cdot) = 0$, and enumerate all size j subsets of $\tilde{O}(v)$. Let S be one such subset of j vertices. If $|\cap_{u \in S} I(u)| \geq i$ then we have detected an (i, j) clique. We remove the fan and the center vertices of this clique from the graph. If the graph is not entirely stored in main memory, the algorithm has to access the disk for every retrieval of the list of predecessors of a vertex of $O(v)$. Once the exploration of a vertex has been completed, the algorithm moves to consider another fan vertex.

In our streaming implementation, the graph is stored on secondary memory in a number of blocks. Every block b, $b = 1, ..., \lceil N/B \rceil$, contains the list of successors and the list of predecessors of B vertices of the graph. Denote by $b(v)$ the block containing vertex v. We denote by $B(b)$ the vertices of block b. We start by analyzing the fan vertices from the first block and proceed until the exploration of the graph reaches the last block. The block currently under examination is moved to main memory. Once the last block has been examined, the exploration continues from the first block. Within a block, we analyze fan vertices from the first vertex to the last vertex of the block.

We start the analysis of a vertex v when block $b(v)$ is moved to main memory for the first time. We then enumerate and recognize cliques formed by all the subsets S of $\tilde{O}(v)$ formed by vertices of block $b(v)$. However, we also have to deal with subsets of $\tilde{O}(v)$ containing vertices not stored in $b(v)$. For this purpose, we store $\tilde{O}(v)$ and the lists of predecessors of the vertices of $\tilde{O}(v) \cap B(b)$ into an auxiliary file $A(b')$ associated with the next next block b' that will be examined in the order, and that contains a vertex of $\tilde{O}(v)$. More precisely, we keep a buffer in main memory for every block b to store information related to those vertices whose exploration will continue from vertex v. Once the buffer reaches a given size, this is moved to the corresponding auxiliary file $A(b)$. In the following we abuse notation by denoting with $A(b)$ also the set of fan vertices v whose exploration will continue with block b.

When a block b is moved to main memory, we first seek to continue the exploration from the vertices of $A(b)$. If the exploration of a vertex v in $A(b)$ cannot be completed within block b, $\tilde{O}(v)$ and the corresponding lists of predecessors collected from block $b(v)$ to block b are stored into the auxiliary file of the next block b' containing a vertex of $\tilde{O}(v)$. We then move to analyze the vertices $B(b)$ of the block. We keep on doing the process till all fan and center vertices have been removed from the graph. It is rather simple to see that the the blocks storing the graph are moved to main memory at most twice.

The core algorithm is preceded by two pruning phases. The first phase removes vertices of high degree as prescribed in [16] since the objective is to detect cores of hidden communities. In the second phase we remove vertices that cannot be selected as fans or centers of an (i, j) clique.

Phase I. Remove all fans v with $|O(v)| \geq 50$ and all centers v with $|I(v)| \geq 50$.

Phase II. Pruning Remove all fans v with $|\tilde{O}(v)| < i$ and all centers with $|\tilde{I}(v)| < j$.

When a fan or a center is removed in Phase II, the in-degree or the out-degree of a vertex is also reduced and this can lead to further removal of vertices. Phase II is carried on for few passes till only few vertices are removed. Phases I and II can be easily executed in a streaming fashion as described in [16]. After the pruning phase, the graph of about 200M vertices is reduced to about 120M vertices. About 65 M of the 80M vertices that are pruned belong to the border of the graph, i.e. they have indegree 1 and outdegree 0.

We then describe the algorithm to detect disjoint bipartite cliques.

Phase III.

1. While there is a fan vertex v with $Fan(v) = 0$
2. Move to main memory the next block b the order from block 1 to $\lceil N/B \rceil$. When the last block is reached, the analysis continues from block 1.
3. For every vertex $v \in A(b) \cup B(b)$ such that $|\tilde{O}(v)| \geq j$

    3.1 For every subset S of size j of $\tilde{O}(v)$ containing vertices from $b(v)$ to b and at least one vertex from b:

    3.2 If $|T = \cap_{u \in S} \tilde{I}(u)| \geq i$ then

    3.2.1 output clique $(T[i], S)$

    3.2.2 set $Fan(\cdot) = 1$ for all vertices of $T[i]$

    3.2.3 set $Center(\cdot) = 1$ for all vertices of S

Figure 4 shows the time performance of the algorithm for detecting disjoint bipartite cliques of size $(4, 4)$ on a system with 256 MB. 70 MB are used by the operating system including operating system's cache. We reserve 20 MB for the buffers storing the information to be moved to the auxiliary files. We maintain 2 bit information $Fan(\cdot)$ and $Center(\cdot)$ for every vertex, and store two *long long* pointers to the list of successors and the list of predecessors of every vertex. Every vertex in the list of adjacent vertices requires 1 long. The graph after the pruning has average out/in 8.75. Therefore, on the average, we need about $.25N + B(2 \times 8 + 17.5 \times 4)$ bytes for a graph of N vertices and block size B. For a graph of 50M vertices this results in a block size of 1.68 M vertices. We performed our experiments with a block size of 1M vertices. We can observe the time performance to converge to a linear function for graphs larger than this size.

5.2 PageRank

The computation of PageRank is expressed in Matrix notation as follows. Let N be the number of vertices of the graph and let $n(j)$ be the out-degree of vertex j. Denote by M the square, stochastic matrix whose entry M_{ij} has value $1/n(j)$ if there is a link from vertex j to vertex i. Denote by $[\frac{1}{N}]_{N \times N}$ the square matrix of size $N \times N$ with entries $\frac{1}{N}$. Vector $Rank$ stores the value of PageRank computed for the N vertices. A matrix M' is then derived by adding transition edges of probability $(1 - c)/N$ between every pair of nodes to include the possibility of jumping to a random vertex of the graph:

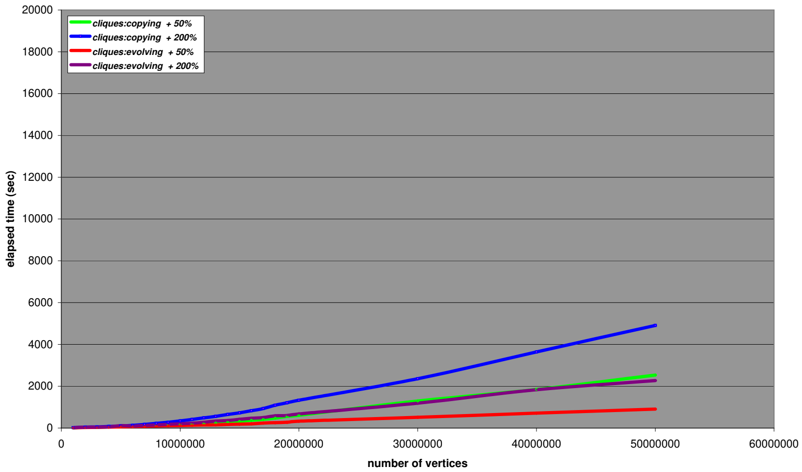

Figure 4: The time performance of the computation of disjoint cliques $(4,4)$

$$M' = cM + (1-c) \times [\frac{1}{N}]_{N \times N}$$

A single iteration of the PageRank algorithm is

$$M' \times Rank = cM \times Rank + (1-c) \times [\frac{1}{N}]_{N \times 1}$$

The most direct approach to implement the computation of PageRank uses a list of successors $Links$, and two arrays $Source$ and $Dest$ that store the vector Rank at iteration i and $i+1$. The computation proceeds until either the error $r = |Source - Dest|$ drops below a fixed value τ or the number of iterations exceed a prescribed value.

We implement the external memory algorithm proposed by Haveliwala [10]. Arrays $Source$ and $Dest$ are partitioned and stored into $\beta = \lceil N/B \rceil$ blocks, each holding the information on B vertices. $Links$ is also partitioned into β blocks. $Links_l$, $l = 0, ..., \beta - 1$, contains all links directed to vertices in block l, i.e. in the range $[lB, (l+1)B - 1]$. We also denote by $d_l(j)$ the number of successors of vertex j in block l.

The external memory algorithm proceeds as follows:

1. $\forall j, Source[j] = 1/N$

2. $r = \tau + 1$

3. while($r > \tau$)

   3.1 For $l = 0, \ldots, \beta - 1$

      3.1.1 $\forall d : Dest_l[d] = 0$

      3.1.2 while(not $Links_l.eof()$)

3.1.2.1 $Links_l.read(j, dest_1, dest_2, \ldots, dest_{d_l(j)})$

3.1.2.2 for $i = 1, \ldots, d_l(j)$, $Dest_l[dest_i] = Dest_l[dest_i] + Source[j]/d(j)$

3.2 $\forall d : Dest_l[d] = c * Dest_l[d] + (1 - c)/N$ /* dampening */

3.3 Write $Dest_l$ to disk

3.5 Normalize($Dest$) /* set $|Dest| = 1$ */

3.6 $r = |Source - Dest|$ /* recompute only every few iterations */

3.7 $Source = Dest$

Since $Links_l$ is sorted on the source field, every pass on $Links_l$ requires only one sequential pass on array $Source$. Step 3.5 is also easily performed in streaming fashion. This algorithm can easily be shown to be I/O optimal.

The time performance of the execution of the algorithm on our synthetic benchmark is shown in Figure 5. For the computation of PageRank we keep in main memory one float for every vertex in the current block. We are then able to store the 50M vertices with 256MB of main memory. The algorithm works in a streaming fashion for which only a small buffer area is required to store the edges.

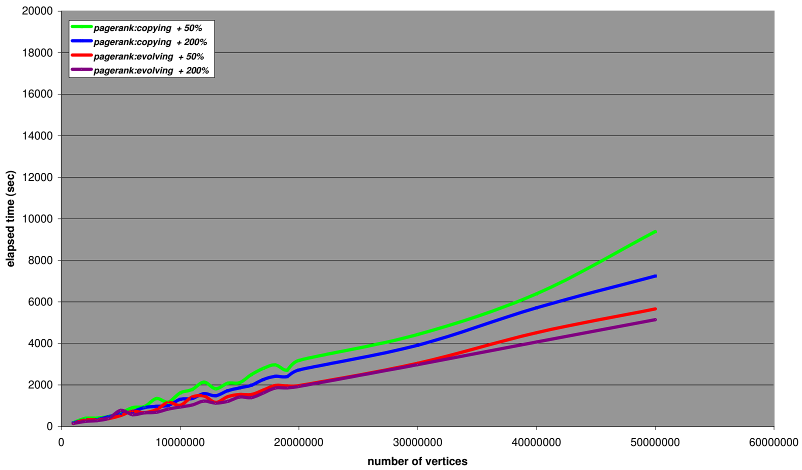

Figure 5: The time performance of the computation of PageRank

5.3 Strongly connected components

It is a well-known fact that SCC can be computed in linear time by two rounds of depth-first search (DFS) [23]. A tree is a DFS tree of a graph if and only if there are no forward cross edges among the non-tree edges of the graph. Unfortunately, so far there are no worst-case efficient external-memory algorithms to compute DFS trees for general directed graphs. The best semi-external approach known still boils down to standard DFS where I/O is restricted to fetching adjacency lists. For graphs with N nodes and M edges this results in $\Theta(N + M/B)$ I/Os.

We therefore apply a recently proposed heuristic for semi-external DFS [22]. It maintains a tentative forest which is modified by I/O-efficiently scanning non-tree edges so as to reduce the number of cross edges. However, this idea does not easily lead to a good algorithm: algorithms of this kind may continue to consider all non-tree edges without making (much) progress. The heuristic overcomes these problems to a large extent by:

- initially constructing a forest with a close to minimal number of trees;
- only replacing an edge in the tentative forest if necessary;
- rearranging the branches of the tentative forest, so that it grows deep faster (as a consequence, from among the many correct DFS forests, the heuristic finds a relatively deep one);
- after considering all edges once, determining as many nodes as possible that have reached their final position in the forest and reducing the set of graph and tree edges accordingly.

The respective DFS program was highly tuned in order to save internal space. Sometimes data is recomputed, sometimes it is temporarily written on file. The used version of the program accesses at most three integer arrays of size N at the same time plus three boolean arrays. With four bytes per integer and one bit for each boolean, this means that the program has an internal memory requirement of $12.375 \cdot N$ bytes. The standard dfs needs to store $16 \cdot \text{av}_\text{d}\text{egree} \cdot N$ bytes or less if if one does not store both endpoints for every edge. Therefore, under memory limitations, standard dfs starts paging at a point when the semi-external approach still performs fine.

Besides the internal computing time, the most important aspect of the heuristic's performance is the number of edges scanned; in [22] a number of different graph classes were tested; most of them had been artificially generated. The classes included random graphs, random geometric graphs, grid graphs, graphs containing cycles and stars, and special random graphs modelling the Webgraph. Real-world data has been processed as well in the form of a telephone call-graph of ATT. Even though one can construct inputs where the heuristic would cause at least as much I/O as the standard DFS approach, all tested instances required the scanning of at most $10 \cdot (N + M)$ edges. Figure 6 shows the time performance of the algorithm when applied to graphs generated according to the EN and the Copying model.

5.4 Algorithms for generating massive Webgraphs

In this section we present algorithms to generate massive Webgraphs. We consider the Evolving Network model and the Copying model. When generating a graph according to a specific model, we fix in advance the number of nodes N of the simulation. The outcome of the process is a graph stored in secondary memory as list of successors.

Evolving Network model. For the EN model we need to generate the end-point of an edge with probability proportional to the in-degree of a vertex [2]. The straightforward approach is to keep in main memory a N-element array $i[]$ where we store the in-degree for each generated node, so that $i[k] = indegree(v_k) + 1$ (the plus 1 is necessary to give to every vertex an initial non-zero probability to be chosen as end-point).

We denote by g the number of vertices generated so far and by I the total in-degree of the vertices $v_1 \dots v_g$ plus g, i.e. $I = \sum_{j=1}^{g} i[j]$. We randomly (and uniformly) generate a number r in the interval $(1 \dots I)$; then, we search for the smallest integer k such that:

[2]The problem of generating efficiently random discrete variables according to general distributions has been for instance considered by Walker [24]; see also Knuth [14]

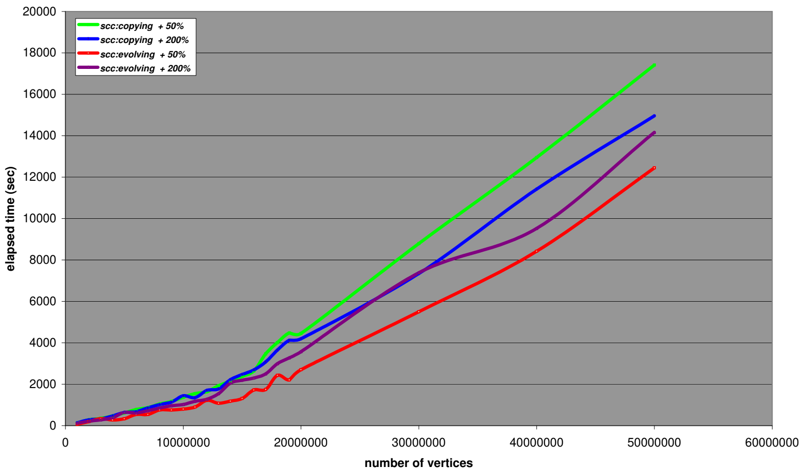

Figure 6: The time performance of the computation of SCCs

$$r \leq \sum_{j=1}^{k} i[j] \tag{1}$$

For massive graphs, this approach has two main drawbacks:

1. We need to keep in main memory the whole in-degree array to speed up operations;
2. We need to quickly identify the vertex v_k for which condition (1) holds.

To overcome both problems we partition the set of vertices in $\sqrt{N}$ blocks. Every entry of a $\sqrt{n}$-element array S contains the sum of the $i[]$ values of a block, i.e. $S[l]$ contains the sum of the elements in the range $i[l\lceil\sqrt{N}\rceil + 1] \ldots i[(l+1)\cdot\lceil\sqrt{N}\rceil]$. To identify in which block the end-point of an edge is, we need to compute the smallest k' such that

$$r \leq \sum_{j=1}^{k'} S[j]$$

The algorithm works by alternating the following 2 phases:

Phase I. We store in main memory tuples corresponding to pending edges, i.e. edges that have been decided but not yet stored. Tuple $t = < g, k', r - \sum_{j=1}^{k'-1} S[j] >$ associated with vertex g, maintains the block number k' and the relative position of the endpoint within the block. We also group together the tuples referring to a specific block. We switch to phase II when a sufficiently large number of tuples has been generated.

Phase II. In this phase we generate the edges and we update the information on disk. This is done by considering, in order, all the tuples that refer to a single block when this is moved to main memory.

For every tuple, we find the pointed node and we update the information stored in $i[]$. The list of successors is also stored as the graph is generated.

In the real implementation we use multiple levels of blocks, instead of only one, in order to speed up the process of finding the endpoint of an edge. An alternative is the use of additional data structures to speed up the process of identifying the position of the node inside the block.

Copying Linear model

The Copying Linear model is parameterized with a copying factor α. Every new vertex u inserted in the graph by the Copying Linear model is connected with d edges to previously existing edges. A random prototype vertex p is selected or vertex u. The endpoint of the lth outgoing edge of vertex u, $l = 1, \ldots, d$, is copied with probability α from the endpoint of the lth outgoing link of vertex p, is chosen uniformly at random among the existing nodes with probability $1 - \alpha$.

A natural strategy would be to generate the graph with a batch process that, alternately, i) generates edges and writes them to disk and ii) reads from disk the edges that need to be "copied". This clearly requires an access to disk for every newly generated vertex.

In the following we present an I/O optimal algorithm that does not need to access the disk to obtain the list of successors of the prototype vertex. We generate for every node $1 + 2 \cdot d$ random integers: one for the choice of the prototype vertex, d for the endpoints chosen at random, and d for the values of α drawn for the d edges. We store the seed of the random number generator at fixed steps, say every x generated nodes.

When we need to copy an edge from a prototype vertex p, we step back to the last time when the seed has been saved before vertex p has been generated, and let the computation progress until the outgoing edges of p are recomputed; for an appropriate choice of x, this sequence of computations is still faster than accessing the disk. Observe that p might also have copied some of its edges. In this case we recursively refer to the prototype vertex of p. We store the generated edges in a memory buffer and write it to disk when complete.

6 Conclusions

In this work we have presented algorithms and experiments for the Webgraph. We plan to carry on these experiments on more recent crawls of the Webgraph in order to assess the temporal evolution of its topological properties. We will also try to get access to the Alexa sample [16] on it our algorithm for disjoint bipartite cliques. Several challenging algorithmic issues are opened in this area. We mention to devise algorithms and data structures for monitoring the dynamic of the Web, network optimization problems on Web and Internet-like graphs. Moreover, the theoretical analysis of stochastic graph models for the Webgraph, that has recently concentrated on the degree of the graph, should be extended to consider a larger set of topological properties.

Acknowledgements: We are very thankful to the Web Base project at Stanford and in particular

Gary Wesley for their great cooperation in making the Web Base crawl available for our study. We also thank James Abello, Guido Caldarelli, Paolo De Los Rios, Camil Demetrescu and Alessandro Vespignani for several helpful discussions.

References

[1] W Aiello, F Chung, and L Lu. A random graph model for massive graphs. In *Proc. ACM Symp. on Theory of computing*, pages 171–180, 2000.

[2] R. Albert, H. Jeong, and A.L. Barabasi. *Nature*, (401):130, 1999.

[3] A.L. Barabasi and A. Albert. Emergence of scaling in random networks. *Science*, (286):509, 1999.

[4] Oliver Riordan Béla Bollobas. Robustness and ulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2003.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engines. In *Proceedings of the 7th WWW Conference*, 1998.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, S. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th WWW conference*, 2000.

[7] C. Cooper and A. Frieze. A general model of undirected web graphs. In *Proc. of the 9th Annual European Symposium on Algorithms(ESA).*

[8] Cyvellance. `www.cyvellance.com`.

[9] P. Erdös and Renyi R. *Publ. Math. Inst. Hung. Acad. Sci*, 5, 1960.

[10] Taher H. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, 1999.

[11] P.M. Pardalos J. Abello and M.G.C. Resen. On maximum cliques problems in very large graphs,external memory algorithms. *Science*, pages 119–130, 1999.

[12] J. Kleinberg. The small world phenomenon: an algorithmic perspective.

[13] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1997.

[14] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[15] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Proc. of 41st FOCS*, 2000.

[16] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber communities. In *Proc. of the 8th WWW Conference*, pages 403–416, 1999.

[17] L. Laura, S. Leonardi, G. Caldarelli, and P. De Los Rios. A multi-layer model for the webgraph. In *On-line proceedings of the* 2nd International Workshop on Web Dynamics., 2002.

[18] L. Laura, S. Leonardi, and S. Millozzi. A software library for generating and measuring massive webgraphs. Technical Report 05-03, DIS - University of Rome La Sapienza, 2003.

[19] Michael Mitznmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2), 2003.

[20] G. Pandurangan, P. Raghavan, and E. Upfal. Using pagerank to characterize web structure. In *Proc. of the 8th Annual International Conference on Combinatorics and Computing (COCOON).*

[21] D.M. Pennock, G.W. Flake, S. Lawrence, E.J. Glover, and C.L. Giles. Winners don't take all: Characterizing the competition for links on the web. *Proc. of the National Academy of Sciences*, 99(8):5207–5211, April 2002.

[22] J.F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002.

[23] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[24] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Mathematical Software*, 3(3):253–256, 1977.

[25] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, (393):440, 1998.

[26] The stanford webbase project. `http://www-diglib.stanford.edu/~testbed/doc2/WebBase/`.

A Appendix

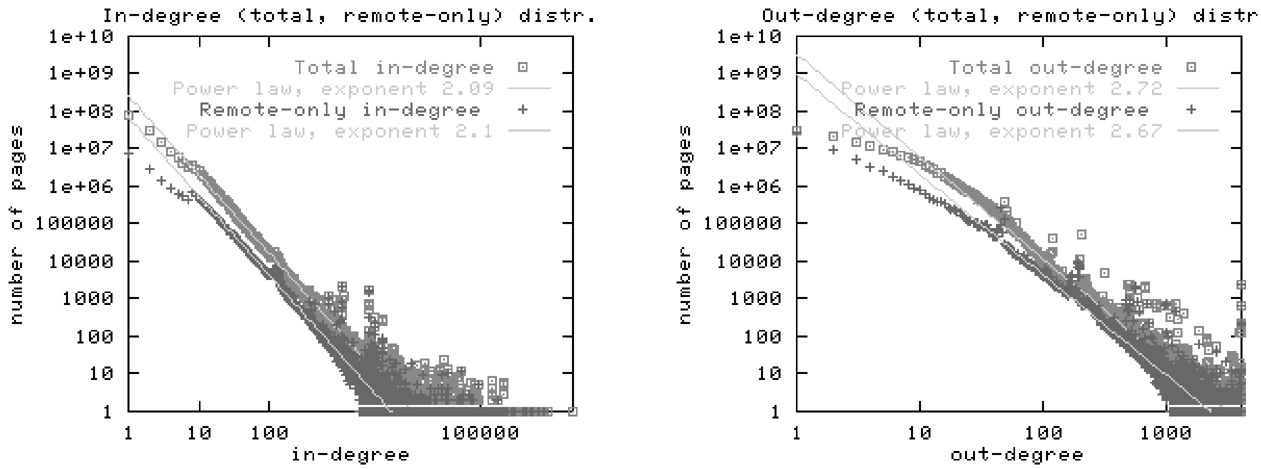

Figure 7: In-degree and Out-degree distribution of nodes of the Webgraph

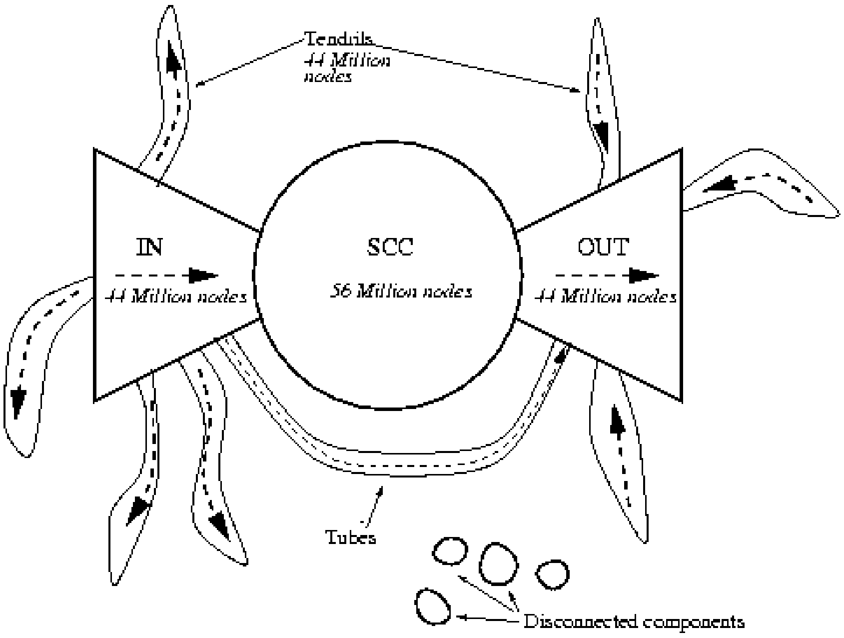

Figure 8: The *bow-tie* shape of the Webgraph

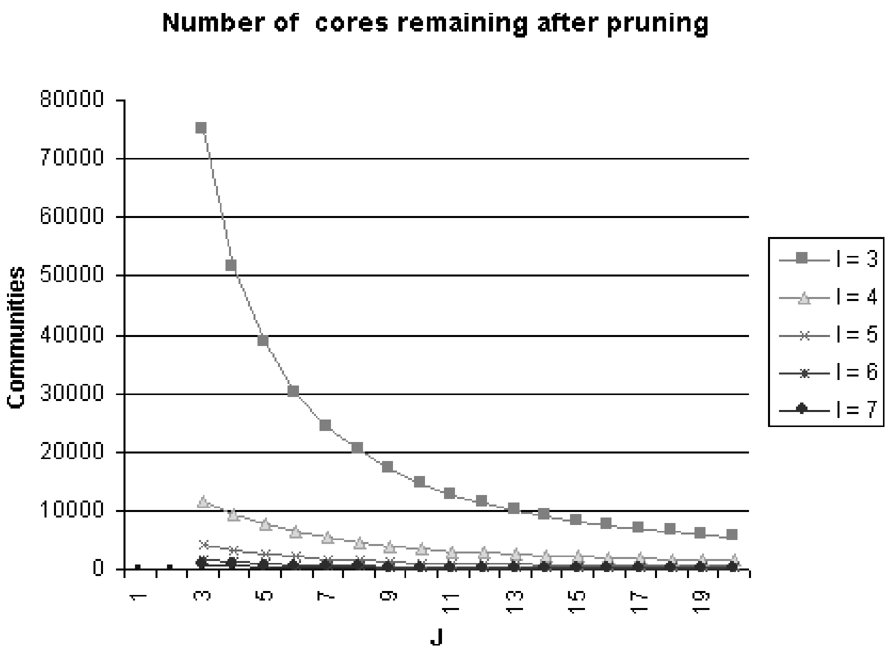

Figure 9: The number of bipartite cliques (i, j) in the Alexa crawl

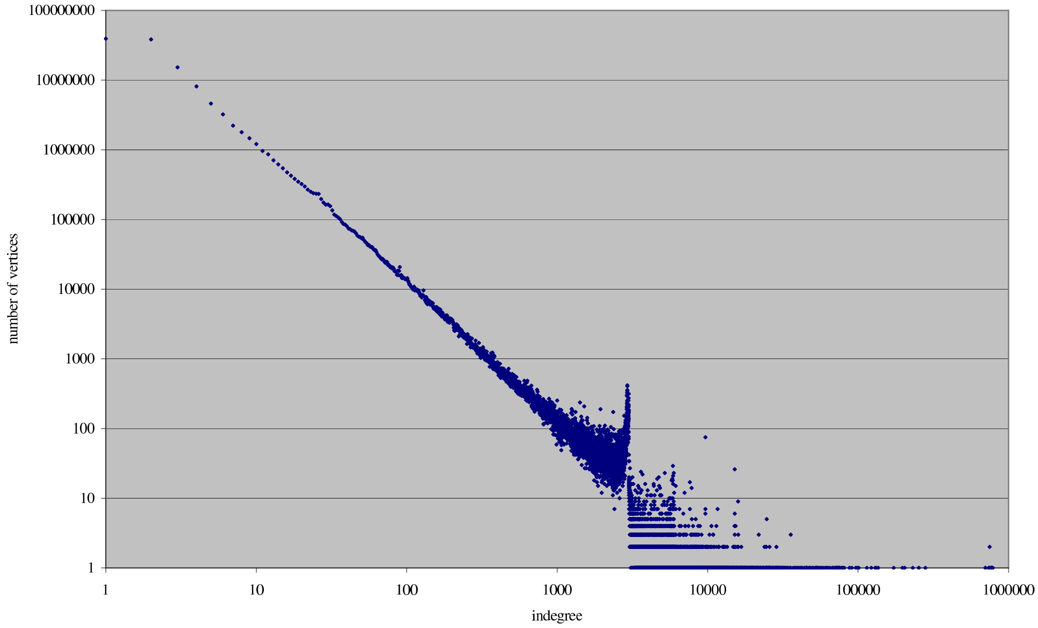

Figure 10: In-degree distribution of the Web Base crawl

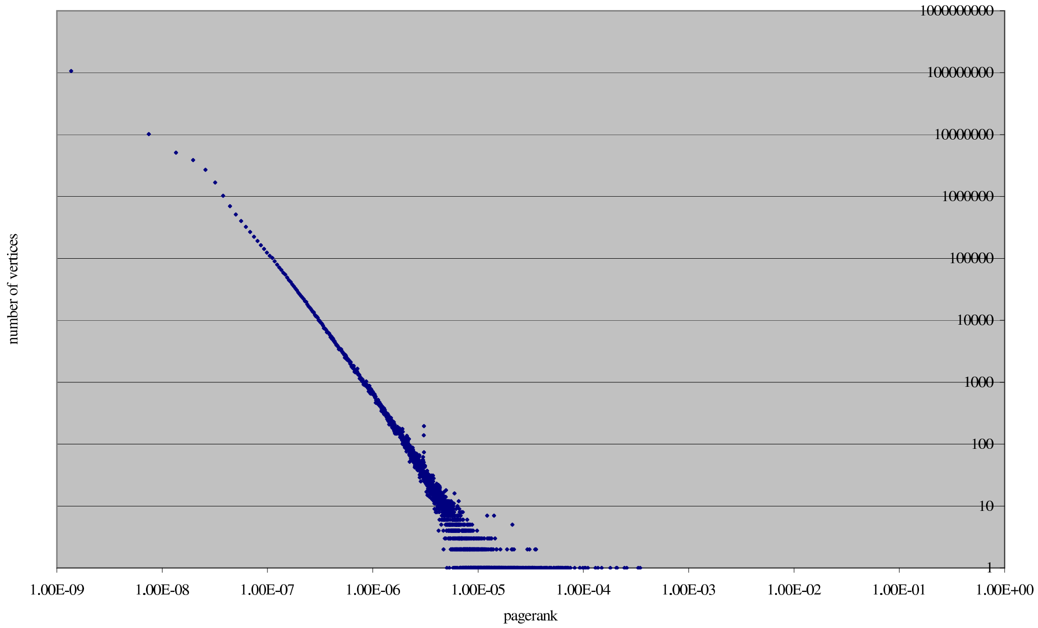

Figure 11: The Page-rank distribution of the Web Base crawl.

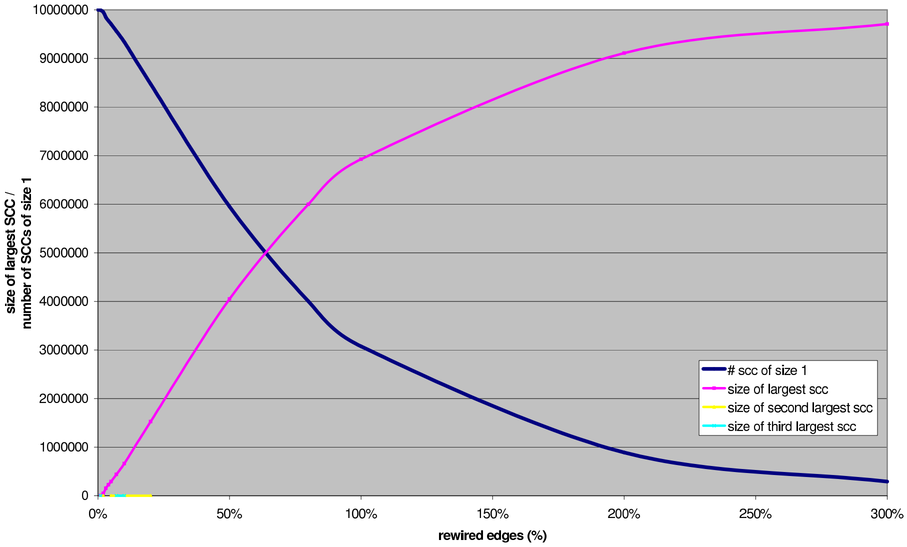

Figure 12: Number and size of SCCs – (Evolving Network Model)