Algorithms and Experiments for the Webgraph*

Luigi Laura[1], Stefano Leonardi[1], Stefano Millozzi[1], Ulrich Meyer[2], and Jop F. Sibeyn[3]

[1] Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", Via Salaria 113, 00198 Roma Italy. {laura,leon,millozzi}@dis.uniroma1.it
[2] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. umeyer@mpi-sb.mpg.de
[3] Halle University, Institute of Computer Science, Von-Seckendorff-Platz 1, 06120 Halle Germany. jopsi@informatik.uni-halle.de

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 [19], 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.

1 Introduction

The *Webgraph* is the graph whose nodes are (static) web pages and edges are (directed) hyperlinks among them. The *Webgraph* has been the subject of a large interest in the scientific community. The reason of such large interest is primarily given to search engine technologies. Remarkable examples are the algorithms for ranking pages such as PageRank [4] and HITS [9].

A large amount of research has recently been focused 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 capture the properties of 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.

* 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 Network" and IST-1999-14186 ALCOM-FT "Algorithms and Complexity in Future Technologies", and by the Italian research project ALINWEB: "Algoritmica per Internet e per il Web", MIUR – Programmi di Ricerca di Rilevante Interesse Nazionale.

G. Di Battista and U. Zwick (Eds.): ESA 2003, LNCS 2832, pp. 703–714, 2003.

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 [11] suggested that the in-degree of the Webgraph follow a *power-law* distribution. Later experiments by Broder et al. [5] on a crawl of 200M pages from 1999 by Altavista confirmed it as a basic property: the probability that the in-degree of a vertex is i is distributed as $Pr_u[\text{in-degree(u)}= i] \propto 1/i^{\gamma}$, for $\gamma \approx 2.1$. In [5] the out-degree of a vertex was also shown to be distributed according to a power law with exponent roughly equal to 2.7 with exception of the initial segment of the distribution. The number of edges observed in the several samples of the Webgraph is about equal to 7 times the number of vertices.

Broder et. al. [5] also presented a fascinating picture of the Web's macroscopic structure: a *bow-tie* shape with a core made by a large strongly connected component (SCC) of about 28% of the vertices. A surprising number of specific topological structures such as bipartite cliques of relatively small size has been observed in [11]. The study of such structures is aimed to trace the emergence of hidden *cyber-communities*. A bipartite clique is interpreted as a core of such a community, defined by a set of fans, each fan pointing to a set of centers/authorities for a given subject, and a set of centers, each pointed by all the fans. Over 100,000 such communities have been recognized [11] on a sample of 200M pages on a crawl from Alexa of 1997.

The Google search engine is based on the popular PageRank algorithm first introduced by Brin and Page [4]. 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. The correlation between the distribution of PageRank and in-degree has been recently studied in a work of Pandurangan, Raghavan and Upfal [15]. 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 it is observed very little correlation between these quantities, i.e., pages with high in-degree may have low PageRank.

The topological properties observed in the WebGraph, as for instance the in-degree distribution, cannot be found in the traditional random graph model of Erdös and Rényi (ER) [7]. Moreover, the ER model is a static model, while the Webgraph evolves over time when new pages are published or are removed from the Web.

Albert, Barabasi and Jeong [1] 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 a constant number of previously inserted vertices chosen according to the *preferential attachment* rule, i.e. with probability proportional to the 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. [10] to explain other relevant properties observed in the Webgraph. For every new vertex entering the graph a prototype vertex p it is selected at random. 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 a link is either copied with probability α from a 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*[10] version of this model, and we refer to it simply as the *Copying* model.

More models of the Webgraph are presented by Pennock et al. [16], Caldarelli et al. [12], Panduragan, Raghavan and Upfal [15], Cooper and Frieze [6]. Mitzenmacher [14] presents an excellent survey of generative models for power-law distributions. Bollobás and Riordan [2] study vulnerability and robustness of scale-free random graphs. Most of the models presented in the literature generate graphs without cycles. Albert et al. [1] amongst others proposed to rewire part of the edges introduced in previous steps to induce links in the graphs.

Outline of the paper. 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 WebBase project at Stanford [19] and made available for our study. The experimental findings on the structure of the WebBase crawl are presented in Section 2.

We also report new properties of some stochastic graph models for the Webgraph presented in the literature. In particular, in Section 3, we study the distribution of the size and of the number of strongly connected components.

This work has required the development of semi-external memory [18] algorithms for computing disjoint bipartite cliques of small size, external memory algorithms [18] based on the ideas of [8] for computing PageRank, and the large scale simulation of stochastic graph models. Moreover, we use the semi-external algorithm developed in [17] for computing Strongly Connected Components. The algorithms and the experimental evaluation of their time performances are presented in Section 4. A detailed description of the software tools developed within this project can be found [13].

2 Analysis of the WebBase Crawl

We conducted our experiments on a $200M$ nodes crawl collected from the WebBase project at Stanford [19] in 2001.

The in-degree distribution follows a power law with $\gamma = 2.1$. This confirms the observations done on the crawl of 1997 from Alexa [11], the crawl of 1999 from Altavista [5] and the notredame.edu domain [3].

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, even for the final segment of the distribution. A deviation from a power law for the initial segment of the distribution was already observed in the Altavista crawl [5].

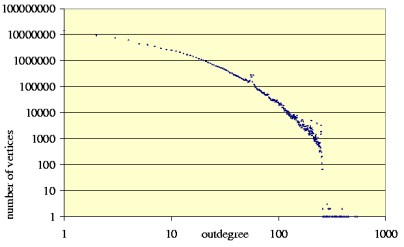

Fig. 1. Out-degree distribution of the Web Base crawl

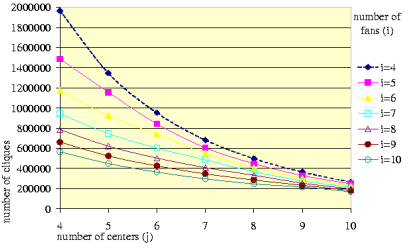

Fig. 2. The number of bipartite cliques (i, j) in the Web Base crawl

We computed the PageRank distribution of the WebBase crawl. Here, we confirm the observation of [15] by showing this quantity 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 $-5.2E-6$, on a range of variation in $[-1, 1]$ from negative to positive correlation. This confirms on much larger scale the observation done by [15] on the brown.edu domain of 100,000 pages that the correlation between the two measures is not significant.

In Figure 2 the graphic of the 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. [11] for the 200M crawl by Alexa. However, 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. A second possible explanation is that our algorithm for finding disjoint bipartite cliques, which is explained in 4.1, is more efficient than the one implemented in [11].

3 Strongly Connected Components

Broder et al. [5] identified a very large strongly connected component of 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 adding edges 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. This is due to the existence of a number of vertices of high in-degree in a graph with in-degree distributed according to a power law. Similar conclusions are also formally obtained for scale-free undirected graphs by Bollobas and Riordan [2]. 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. We observe the size of the largest SCC to increase smoothly with the number of edges that are rewired up to span a big part of the graph. We also observe that the number of SCCs decreases smoothly with the increase of the percentage of rewired edges. This can be observed in Figure 3 for the Copying model on a graph of 10M vertices. A similar phenomenon is observed for the Evolving Network model.

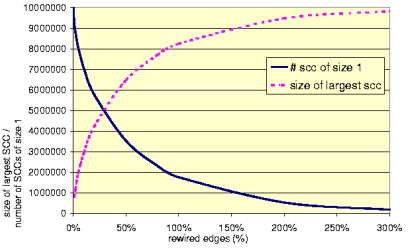

Fig. 3. Number and size of SCCs – (Copying Model)

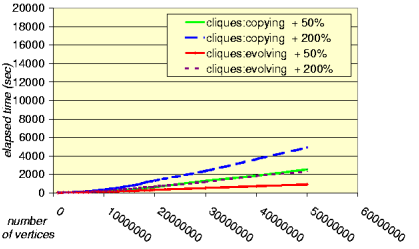

Fig. 4. The time performance of the computation of disjoint cliques $(4, 4)$

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 [17]. This algorithm together with its time performance is described in Section 4.

4 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 behavior 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 256MB 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.

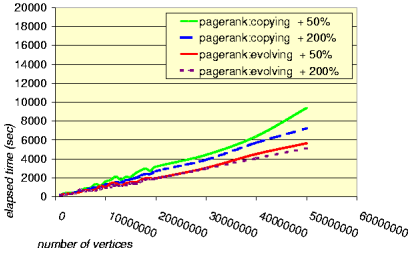

Fig. 5. The time performance of the computation of PageRank

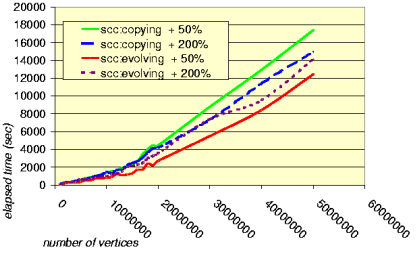

Fig. 6. The time performance of the computation of SCCs

4.1 Disjoint Bipartite Cliques

In [11] 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. The algorithm proposed by Kumar et al. [11] 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 $Fan(\cdot) = 0$, and let $\tilde{O}(v)$ 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.

We first outline 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 semi-external 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, and by $B(b)$ the vertices of block b. We start by analyzing the fan vertices from the first block and proceed until 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.

We start the analysis of a vertex v when block $b(v)$ is moved to main memory for the first time. We start considering all subsets S of $\tilde{O}(v)$ formed by vertices of block $b(v)$. However, we also have to consider those subsets of $\tilde{O}(v)$ containing vertices of other blocks, for which the list of predecessors is not available in main memory. For this purpose, consider the next block b' that will be examined that contains a vertex of $\tilde{O}(v)$. 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 vertex b'. We actually buffer the access to the auxiliary files. Once the buffer of block b 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, the list of predecessors of the vertices of $\tilde{O}(v)$ in blocks from $b(v)$ 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 this till all fan and center vertices have been removed from the graph. It is rather simple to see that every block is 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 suggested in [11] since the objective is to detect cores of hidden communities. In a 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. 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 few times till only few vertices are removed. Phases I and II can be easily executed in a streaming fashion as described in [11]. After the pruning phase, the graph of about 200M vertices is reduced to about 120M vertices. About 65M of the 80M vertices that are pruned belong to the border of the graph, i.e. they have in-degree 1 and out-degree 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 to be examined.
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 $\tilde{O}(v)$, with the list of predecessors of vertices in S stored either in the auxiliary file $A(b)$ or in block 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 20MB for the buffers of the auxiliary files. We maintain 2 bit information $Fan(\cdot)$ and $Center(\cdot)$ for every vertex, and store two 8bytes pointer to the list of successors and the list of predecessors of every vertex. Every vertex in the list of adjacent vertices requires 4 bytes. The graph after the pruning has average out/in 8.75. Therefore, on the average, we need about $0.25N + B(2 \times 8 + 17.5 \times 4)$ bytes for a graph of N vertices and block size B. For a graph of 50M vertices this results in a block size of 1.68M 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.

4.2 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 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:

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

A single iteration of the PageRank algorithm is

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

We implement the external memory algorithm proposed by Haveliwala [8]. The algorithm 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.

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, where $Links_l$, $l = 0, ..., \beta - 1$, contains for every vertex of the graph only those successors directed to vertices in block l, i.e. in the range $[lB, (l+1)B-1]$. We bring to main memory one block of *Dest* per time. Say we have the *ith* block of *Dest* in main memory. To compute the new PageRank values for all the nodes of the ith block we read, in a streaming fashion, both array *Source* and $Links_i$. From array *Source* we read previous Pagerank values, while from $Links_i$ we have the list of successors (and the out-degree) for each node of the graph to vertices of block i, and these are, from the above Pagerank formula, exactly all the information required.

The main memory occupation is limited to one float for each node in the block, and, in our experiments, 256MB allowed us to keep the whole *Dest* in memory for a 50M vertices graph. Only a small buffer area is required to store *Source* and *Links*, since they are read in a streaming fashion. The time performance of the execution of the algorithm on our synthetic benchmark is shown in Figure 5.

4.3 Strongly Connected Components

It is a well-known fact that SCCs can be computed in linear time by two rounds of depth-first search (DFS). Unfortunately, so far there are no worst-case efficient external-memory algorithms to compute DFS trees for general directed graphs. We therefore apply a recently proposed heuristic for semi-external DFS [17]. 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 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 $16 \cdot$

$avg - degree \cdot N$ bytes or less 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. Figure 6 shows the time performance of the algorithm when applied to graphs generated according to the EN and the Copying model.

4.4 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. 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 $r \leq \sum_{j=1}^{k} i[j]$. For massive graphs, this approach has two main drawbacks: i.) We need to keep in main memory the whole in-degree array to speed up operations; ii.) We need to quickly identify the integer k.

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 $i[l\lceil\sqrt{N}\rceil+1] \dots i[(l+1) \cdot \lceil\sqrt{N}\rceil]$. 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 model. The Copying model is parameterized with a copying factor α. Every new vertex u inserted in the graph by the Copying model is connected with d edges to previously existing vertices. A random prototype vertex p is also selected. The endpoint of the lth outgoing edge of vertex u, $l = 1, \ldots, d$, is either copied with probability α from the endpoint of the lth outgoing link of vertex p, or 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.

5 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 [11] and execute on it our algorithm for disjoint bipartite cliques.

Acknowledgments. We are very thankful to the WebBase project at Stanford and in particular Gary Wesley for their great cooperation. We also thank James Abello, Guido Caldarelli, Paolo De Los Rios, Camil Demetrescu and Alessandro Vespignani for several helpful discussions. We also thanks the anonymous referees for many valuable suggestions.

References

1. R. Albert, H. Jeong, and A.L. Barabasi. *Nature*, (401):130, 1999.
2. O. Riordan B. Bollobas. Robustness and ulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2003.
3. A.L. Barabasi and A. Albert. Emergence of scaling in random networks. *Science*, (286):509, 1999.
4. 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.
5. 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.
6. C. Cooper and A. Frieze. A general model of undirected web graphs. In *Proc. of the 9th Annual European Symposium on Algorithms(ESA)*.
7. P. Erdös and Renyi R. *Publ. Math. Inst. Hung. Acad. Sci*, 5, 1960.
8. T. H. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, 1999.
9. J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1997.
10. R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Proc. of 41st FOCS*, pages 57–65, 2000.
11. 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.
12. 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.
13. 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.
14. M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2), 2003.
15. G. Pandurangan, P. Raghavan, and E. Upfal. Using pagerank to characterize web structure. In Springer-Verlag, editor, *Proc. of the 8th Annual International Conference on Combinatorics and Computing (COCOON)*, LNCS 2387, pages 330–339, 2002.
16. 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.
17. 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 (SPAA)*, pages 282–292, 2002.
18. J. Vitter. External memory algorithms. In *Proceedings of the 6th Annual European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.
19. The stanford webbase project. `http://www-diglib.stanford.edu/~testbed/doc2/WebBase/`.