
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 10, no. 2, pp. 219–236 (2006)

Algorithms and Experiments for the Webgraph

This work is dedicated to the memory of Jop F. Sibeyn.

Debora Donato *Luigi Laura*
Stefano Leonardi *Stefano Millozzi*

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
http://www.dis.uniroma1.it/
{donato,laura,leon,millozzi}@dis.uniroma1.it

Ulrich Meyer

Max-Planck-Institut für Informatik,
http://www.mpi-sb.mpg.de
umeyer@mpi-sb.mpg.de

Jop F. Sibeyn

Institute of Computer Science
Halle University

Abstract

In this paper we present an experimental study of the statistical and topological properties of the Webgraph. This work has required the development of a set of external and semi-external algorithms for computing properties of massive graphs, and for the large scale simulation of stochastic graph models. We use these algorithms for running experiments on a large crawl from 2001 of 200M pages and about 1.4 billion edges made available by the WebBase project at Stanford [19], and on synthetic graphs obtained by the large scale simulation of stochastic graph models for the Webgraph.

Article Type	Communicated by	Submitted	Revised
regular paper	D. Wagner	February 2005	February 2006

A preliminary version of this paper has appeared in the Proceedings of the 11th Annual European Symposium on Algorithms (ESA03), pp. 703-714, LNCS 2832, Springer-Verlag, 2003.

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 [10].

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.

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. [12] 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 [12]. 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 [12] on a sample of 200 million 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 it is also observed a very weak 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) [8]. 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. [11] 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* [11] 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. [13], 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 a comprehensive set of external and semi-external memory algorithms for studying statistical and topological properties of massive Webgraphs. We test these algorithms on a crawl of about 200 million pages collected in 2001 by the WebBase project at Stanford [19], a widely used sample for testing Web applications and studying the structure of the Web. The experimental findings on the structure of the WebBase crawl are presented in Section 2.

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 [9] 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 we developed

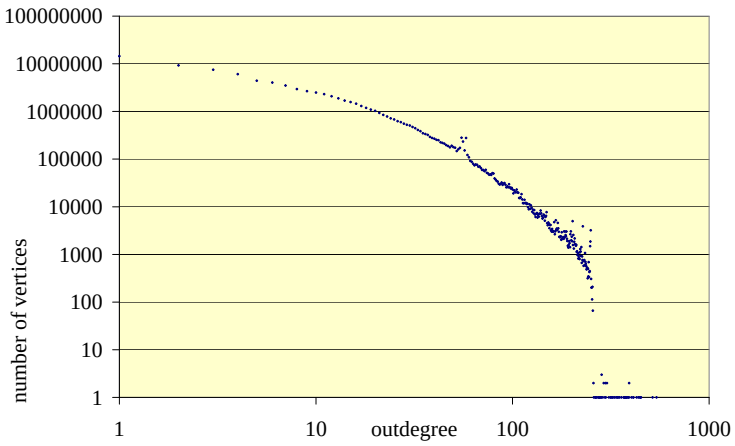

Figure 1: Out-degree distribution of the Web Base crawl

can be found in [7].

2 Analysis of the WebBase crawl

We first present several experimental findings that have been obtained by running our algorithms on a 200 million nodes crawl collected from the WebBase project at Stanford [19] in 2001. The in-degree distribution of the graph follows a power-law with $\gamma = 2.1$, to confirm the results of the initial study of the notredame.edu domain [3] and the observations done on a crawl from 1997, collected by Alexa [12], and a crawl from 1999 done by Altavista [5].

In Figure 1 the out-degree distribution of the WebBase crawl is also 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. The final cut is due to some limitations of the crawler. A deviation from a power-law for the initial segment of the distribution was already observed in the Altavista crawl [5]. The distribution of the out-degree is considered at some extent less meaningful than the in-degree since the outlinks of a page are often all generated by a single content creator. We also computed the correlation (more precisely, we computed the Pearson's correlation coefficient) between in-degree and out-degree. This assumes value 0.022 on a range of variation in $[-1, 1]$ from negative to positive correlation, there showing the two observables almost uncorrelated, i.e., sites with large in-degree are not likely to have large out-degree.

Another important measure is the PageRank of the pages. We computed the

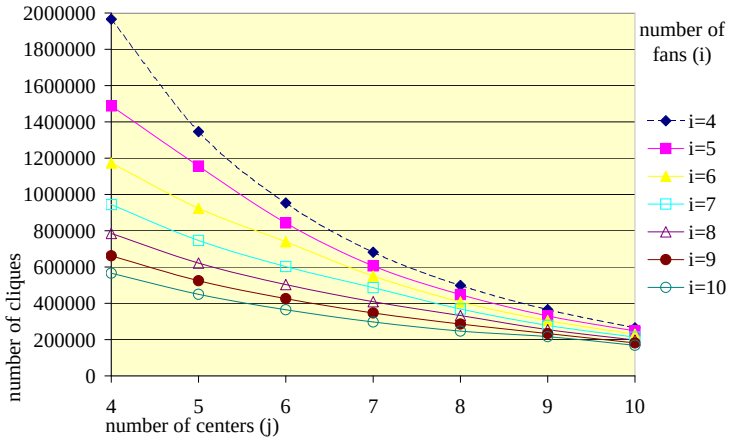

Figure 2: The number of bipartite cliques (i, j) in the Web Base crawl

PageRank distribution of the WebBase crawl. Here, we confirm the observation of [15] by showing that this quantity is distributed according to a power-law with exponent $\gamma = 2.109$. We also computed the correlation between PageRank and in-degree. This assumes a value of 0.307, therefore confirming the observation of [15] on a weak correlation between in-degree and PageRank.

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. [12] for the crawl by Alexa. However, we detect a number of bipartite cliques of size $(4, j)$, with $j = 1, \ldots, 10$, 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.

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 in order to induce cycles by adding in the graph a number of edges ranging from 1% to 300% of the number of vertices. Recall that these graphs contain 7 times as many edges as the vertices.

An interesting observation we make is that, differently from the Erdös-Renyi

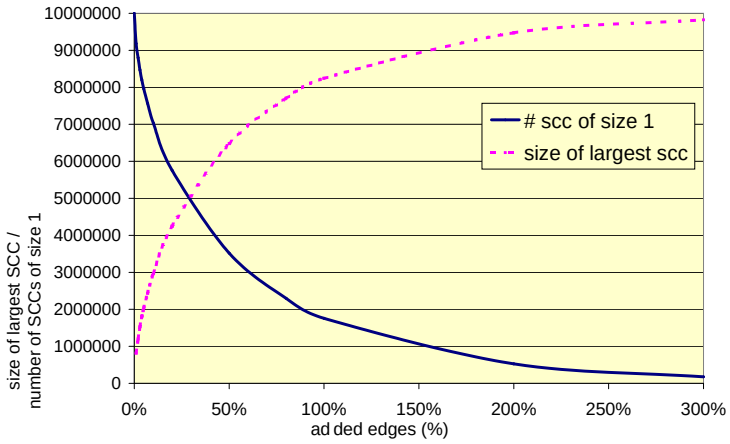

Figure 3: Number and size of SCCs − (Copying Model)

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. What we observe is that the size of the largest SCC increases smoothly with the number of addes edges until it spans a big part of the graph. We also observe that the number of SCCs decreases smoothly with the increase of the percentage of added edges. This can be observed in Figure 3 for the Copying model on a graph of 10 million vertices. A similar phenomenon is observed for the Evolving Network model. This phenomenon, not previously known at the best of our knowledge, deserves a much closer attention and analytical approach for a complete explanation. A similar conclusion has also been obtained analytically for scale-free undirected graphs by Bollobas and Riordan [2] about at the same time this paper has appeared.

Identifying strongly connected components in a graph stored on secondary memory is a non-trivial task. We used for this purpose a semi-external algorithm developed in [17] whose implementation together with its time performance is described in Section 4.

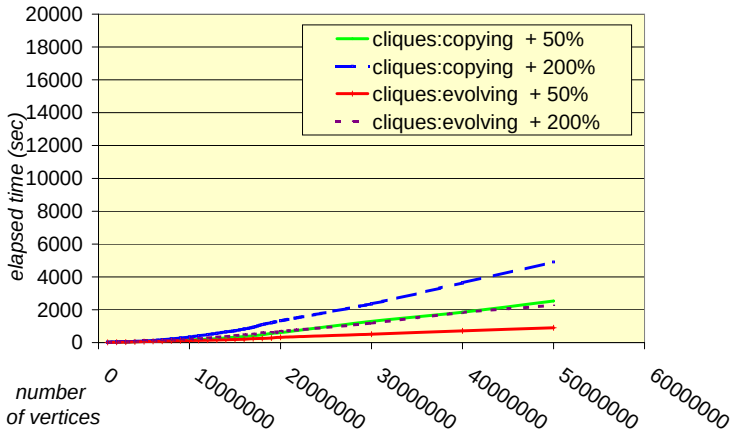

Figure 4: The time performance of the computation of disjoint cliques $(4, 4)$

4 Algorithms for analyzing and generating Web graphs

In this section, we present the external and semi-external memory algorithms we developed for analyzing massive Webgraphs and the study of their time performances. Moreover, we will present some 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 50 million vertices with average degree 7, and later added a number of edges equal to 50% and 200% of the vertices. The presence of cycles is fundamental for both computing SCCs and PageRank. 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 50 million 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.

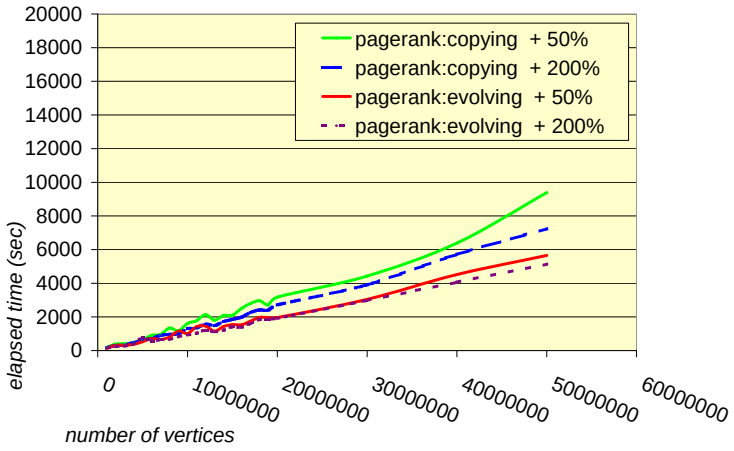

Figure 5: The time performance of the computation of PageRank

4.1 Disjoint bipartite cliques

A bipartite clique (i, j) has i fan vertices all linked to j center vertices. Kumar et al. [12] present a heuristic for enumerating and listing disjoint bipartite cliques (i, j). Disjointness means that all cliques found by the algorithm have disjoint sets of fan and centers, thus not excluding a fan vertex of a first clique being a center vertex of a second clique. A first phase prunes the graph: since the objective is to detect cores of hidden communities, all the vertices with high degree are removed. Then, before enumerating the cliques, we remove from the set of potential fans all the vertices with outdegree smaller than j, and from the list of potential centers all the vertices with indegree smaller than i.

The algorithm of [12] then enumerates all bipartite cliques of the pruned graph and selects a set of disjoint bipartite cliques that form the solution. This enumeration phase is done in main memory, therefore limiting the size of the graphs that can be analyzed by this algorithm.

The novelty of our approach is a semi-external implementation of the algorithm of [12], i.e., the algorithm can process graphs of any size as soon as a fixed amount of information for each vertex of the graph can be stored in main memory. An efficient semi-external implementation of the algorithm requires a number of changes in the last phase of the algorithm that we'll present in the following of this section.

To store the information on nodes that are still potential fans or centers, we use two n-bit arrays Fan and $Center$, stored in main memory. We set $Fan(v) := 0$ ($Center(v) := 0$) whether vertex v is still a potential fan (center),

$Fan(v) := 1$ ($Center(v) := 1$) otherwise. We also denote by

- $I(v)$ the list of predecessors of vertex v in the original graph.
- $O(v)$ the list of successors of vertex v in the original graph.
- $\tilde{I}(v)$ the set of predecessors of vertex v with $Fan(\cdot) = 0$.
- $\tilde{O}(v)$ the set of successors of vertex v with $Center(\cdot) = 0$.

Along the execution of the algorithm we search for fan vertices v having a subset $S \subseteq \tilde{O}(v)$ with $|S| = j$ such that

$$|\cap_{u \in S} I(u)| \geq i. \tag{1}$$

This gives the evidence of a (i, j) clique in the graph. The main difficulty with a graph not completely stored in main memory is the evaluation of expression (1): a distinct access to disk is possibly needed for every vertex of $O(v)$.

To overcome this hurdle, in our semi-external implementation, the graph is stored on secondary memory in a number of blocks, each one containing the list of successors and the list of predecessors of B vertices of the graph. Blocks are numbered $b = 1, ..., \lceil N/B \rceil$. Denote by $b(v)$ the index of the block containing vertex v, and by $B(b)$ the set of vertices of block b. The algorithm will then scan the blocks of the graph a number of times from the first to the last block in search of bipartite cliques until either $Fan(v) = 1, \forall v$, or $Center(v) = 1, \forall v$.

In the following we give a high level description of the algorithm. When a block is moved to main memory, we scrutiny in sequence all vertices of the block. For a generic vertex v, we consider all subsets of cardinality j of $\tilde{O}(v)$. Let S be the generic subset. We can evaluate expression (1) for a set S containing only vertices of block $b(v)$ by simply accessing the information currently stored in main memory. If one such set forms a clique we have done with vertex v. We instead postpone the evaluation for those subsets S containing vertices of other blocks. Let b' be the block that will be loaded sooner in main memory containing 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 block b'. The access to the auxiliary files is actually buffered: file $A(b)$ is written only after the buffer reaches a given size. In the following we abuse notation by denoting with $A(b)$ also the set of fan vertices whose exploration will continue with block b.

When a block b is loaded, we should then decide for both vertices of $A(b)$ and $B(b)$ with $Fan(v) = 0$. We start with vertices of $A(b)$. If the evaluation of a vertex v of $A(b)$ can be completed by looking at the information stored in block b, we have done with this vertex. Otherwise we postpone again evaluation of vertex v to the block that will be loaded sooner containing a vertex of $\tilde{O}(v)$. We then move to consider vertices of $B(b)$ as described in the previous paragraph. We keep on doing this till all fan and center vertices have been removed from

```
// Phase I:
// Pruning the graph
Remove all fans v with |O(v)| ≥ 50 and all centers v with |I(v)| ≥ 50.
repeat
    no_more_to_remove:=true;
    for each vertex v do
        if |Õ(v)| < i or |Ĩ(v)| < j then
            Remove v;
            no_more_to_remove:=false;
        end
    end
until no_more_to_remove;
// Phase II:
// Enumerating the bipartite cliques of the pruned graph
while there is a fan vertex v with Fan(v) = 0 AND a center vertex u
with Center(u) = 0 do
    Load into memory the next block b to be examined;
    for every vertex v ∈ A(b) ∪ B(b) such that |Õ(v)| ≥ j do
        for every subset S of size j of Õ(v) such that the list of
        predecessors of each vertex in S is either stored in A(b) or in block
        b do
            T := ∩_{u∈S} Ĩ(u);
            if |T| ≥ i then
                output clique (T[i], S);
                set Fan(·) := 1 for all vertices of T[i];
                set Center(·) := 1 for all vertices of S;
            end
        end
    end
end
```

Algorithm 1: Bipartite cliques enumeration

the graph. It is rather simple to see that we decide on all vertices by scanning the graph at most twice.

Note that the pruning phase can be easily executed in a streaming fashion as described in [12]. In our experiments, the graph of about 200 million vertices is reduced to about 120M vertices after the pruning. 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.

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 8 bytes 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 an average degree equals to 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. We performed our experiments with a block size of 1 million vertices. We can observe the time performance to converge to a linear function for graphs larger than this size. On the same machine, for comparison, it took around 4 hours to compute the bipartite cliques of the 200 million nodes WebBase sample.

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

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

We implement the external memory algorithm proposed by Haveliwala [9]. The algorithm uses a list of successors $Links$, and two arrays $Source : /u/rt/proj/cvs_rt/rt/ref/jgaa/final/200$ $10/10-02-Leonardi/Donato+2006.10.2.tex, v$ 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 exceeds a prescribed value.

Arrays $Source : /u/rt/proj/cvs_rt/rt/ref/jgaa/final/2006-10/10-02-Leonardi/Donato + 2006.10.2.tex, v$ 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 i-th block of $Dest$ in main memory. To compute the new PageRank values for all the nodes of the i-th block we read, in a streaming fashion, both array $Source : /u/rt/proj/cvs_rt/rt/ref/jgaa/final/2006-10/10-02-Leonardi/Donato + 2006.10.2.tex, v$ and $Links_i$. From array $Source : /u/rt/proj/cvs_rt/rt/ref/jgaa/final/2006-10/10-02-Leonardi/Donato+$

$2006.10.2.tex, v$ we read previous PageRank values ($Rank_{i-1}$), while $Links_i$ represents the colums of the matrix M associated with block i. These are, from the above PageRank formula, exactly all the information required to compute the new PageRank values ($Rank_i$).

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 50 million vertices graph. Only a small buffer area is required to store $Source : /u/rt/proj/cvs_rt/rt/ref/jgaa/final/2006 - 10/10 - 02 - Leonardi/Donato + 2006.10.2.tex, v$ 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. For comparison, it took around 5 hours and a half to compute the PageRank on the 200 million nodes WebBase sample.

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 a number of bytes equal to $16 \cdot N$ times the average degree; this amount can be reduced 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.

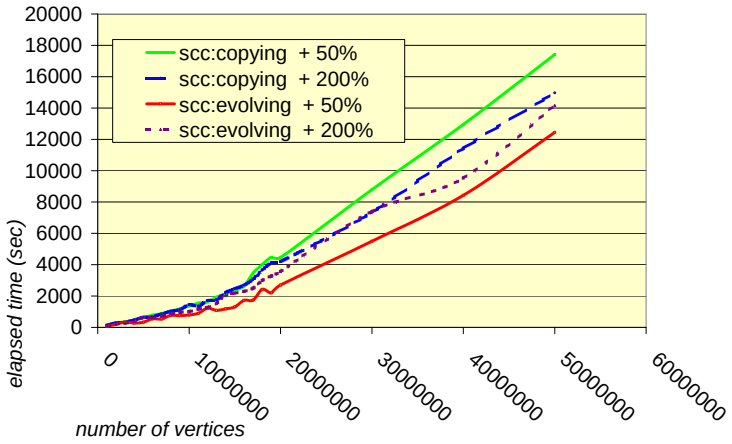

Figure 6: The time performance of the computation of SCCs

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. In Figure 7 we show the time performance of our implementation of these algorithms.

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 $i[l\lceil\sqrt{N}\rceil+1] \dots i[(l+$

Figure 7: The time performance of the generation of Webgraphs

$1) \cdot \lceil\sqrt{N}\rceil]$. 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, e.g. a search tree or a hash table.

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, first generates edges and writes them to disk, and second reads from

```
Data: v,d,α
Result: O(v)
// v:  the index of the node
// d:  the outdegree (and therefore the number of successors
   to be generated)
// α:  the copying probability
// O(v) = O_1(v)...O_d(v):  the list of successors of vertex v;


// RandInt(seed,n) returns a random integer between 1 and n
// Rand01(seed) returns a random number between 0 and 1
seed:=GetSeed(v);
// We first choose the prototype vertex p
p:=RandInt(seed,v-1);
// We compute its successors
O(p)=GenerateSuccessors(p, d, α);
// We generate the successors of d by either copying from p
   or generating them randomly
for i:=1 to d do
   tempvertex:=RandInt(seed,v-1);
   coin:=Rand01(seed);
   if coin < α then
      // We copy
      O_i(v):=O_i(p);
   else
      // We use the randomly generated one
      O_i(v):=tempvertex;
   end
end
```

Procedure `GenerateSuccessors(`*v, d, α*`)`

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 algorithm that does not need to access the disk to obtain the list of successors of the prototype vertex. With $O(v)$ we denote the list of successors of vertex v. The procedure `GenerateSuccessors(v,`d`,`α`)` generates the successors of the vertex v: note that, since we may copy from a prototype vertex p, there is a recursive call to get the successors of p. The main idea is to store the seed of the random number generator at fixed steps, say every x generated nodes. We use the function `GetSeed(i)` that stores, retrieves and, if necessary, computes the value of the seed depending on the index vertex; we know that we generate $1 + 2 \cdot d$ random numbers for every node: One for the choice of the prototype vertex, d for the endpoints chosen at random, and d to decide, against the value of α, if we copy or not.

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.

In our experiments we gave to x all the available memory left after the allocation of the other data stuctures needed by the procedure.

5 Conclusions

In this work we have presented algorithms and experiments for the Webgraph. We designed new algorithms, that we implemented together with known ones, in order to develop a set of routines able to generate and analyze massive graph in secondary memory. More details on this softare library, that is freely available, can be found in [7].

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.

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.

This research has been 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", by the WEB-MINDS project supported by the Italian MIUR under the FIRB program and by the Italian research project ALINWEB: "Algorithmica per Internet e per il Web", MIUR – Programmi di Ricerca di Rilevante Interesse Nazionale.

References

[1] R. Albert, H. Jeong, and A. Barabasi. Diameter of the World Wide Web. *Nature*, (401):130, 1999.

[2] O. R. B. Bollobas. Robustness and vulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2003.

[3] A. 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] D. Donato, L. Laura, S. Leonardi, and S. Millozzi. A software library for generating and measuring massive webgraphs. Technical Report D13, COSIN European Research Project, 2004. `http://www.dis.uniroma1.it/~cosin/html_pages/COSIN-Tools.htm`.

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

[9] T. H. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, 1999.

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

[11] 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.

[12] 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.

[13] 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.

[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. Pennock, G. Flake, S. Lawrence, E. Glover, and C. 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. 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/`.