COSIN

COSIN

IST–2001-33555

COevolution and Self-organization In dynamical Networks

A library of software tools for performing measures on large networks

Deliverable Number:	D13
Delivery Date:	March 2004
Classification:	Public
Partner owning:	**CR2** (UDRLS)
Contact authors:	Debora Donato, Luigi Laura, Stefano Leonardi, Stefano Millozzi.
Project Co-ordinator:	Guido Caldarelli (INFM) *guido.caldarelli@roma1.infn.it*, Istituto Nazionale Fisica per la Materia
Partners:	**CR2** (UDRLS) Università "La Sapienza", *Italy*; **CR3** (UB) Universitat de Barcelona, *Spain*; **CR4** (UNIL) Université de Lausanne, *Switzerland*; **CR5** (ENS) École Normale Supérieure, Paris, *France*; **CR7** (UNIKARL) Universität Karlsruhe, *Germany*; **CR8** (UPSUD) Université de Paris Sud, *France.*

Information Society

Project funded by the European Community under the "Information Society Technologies" Programme (1998-2002)

Abstract

The Webgraph is the graph whose nodes are the (static) HTML pages and the (directed) edges are the hyperlinks between pages. In order to study the statistical and topological properties of the Webgraph and to study models for the Webgraph we need to generate and measure graphs with more than one billion edges. We developed a collection of routine that are able to deal with massive graphs stored in files in secondary memory. In particular, our routines can generate graphs according to many of the models presented in the literature; we also provide programs that can measure these graphs (the ones generated according to known models) as well as real samples of the webgraph. We present a "multifile" format to represent graphs in secondary memory; we include routines that convert some graph file formats from/to our .ips multifile format. Binaries and source code of all the program of this library are freely available; to compile them the gcc compiler version >= 2.9 and linux operating system are needed. The library has been tested with graphs up to 2 billion edges.

Contents

1 Introduction

The Webgraph is the graph whose nodes are the (static) HTML pages and the (directed) edges are the hyperlinks between pages. This has been the subject of extensive attention because of the many applications that benefited from the analysis of the link structure of the Web, primarly Web mining. One example is represented by the algorithms for ranking pages such as Page Rank [2] and HITS [6]. Link analysis is also at the basis of the sociology of content creation, and the detection of structures hidden in the web (such as bipartite cores of cyber communities and webrings [8]).

The experimental study of the statistical and topological properties is at the core of this discipline and at the basis of the validation of stochastic graph models for the Web.

To study and analyze the webgraph we need to deal with massive graph. In this deliverable we present a collection of algorithms and related implementations that are able to generate and measure massive graphs in secondary memory. For further readings concerning properties of the Webgraph, see [4], and a detailed comparison studio over different Webgraph models can be found in [3].

This deliverable is organized as follows: next section presents the external and semi-external memory algorithm we developed in order to generate and analyze Webgraphs. In Section 3 we briefly describe the file formats that we use to represent both the graphs and the results of the measurement processes. Section 4 describe in details our routines. Some examples of the usage are presented in Section 5.

2 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.

2.1 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.

Copying model. The Copying model (Kumar *et al.* [7]) 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, $l = 1, \dots, d$, 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.

Evolving Network model. Albert, Barabasi and Jeong [1] presented this model in which, at every discrete time step, a new vertex is introduced in the graph, and connects to existing vertices with a constant number of edges. A vertex is selected as the end-point of an edge with probability proportional to its in-degree, with an appropriate normalization factor (this is called *preferential attachment*). This model shows a power law distribution over the in-degree of the vertices with exponent roughly -2 when the number od edges that connect every vertex to the graph is 7. 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 $r \le \sum_{j=1}^{k} i[j]$. 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 $r \le \sum_{j=1}^{k'} S[j]$

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 library 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.

2.2 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 [5]. 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.

2.3 Disjoint bipartite cliques

In [8] 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. [8] 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 [8] 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 [8]. 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 $S$
```

2.4 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 [11]. 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 $16 \cdot \mathrm{avg} - \mathrm{degree} \cdot N$ 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.

3 Data representation and multifiles

A first and natural problem when one deals with massive data in secondary memory is the size limit of a single file[1]. All our routine operate on a *multifile*, i.e. we represent a single graph with more than one file. More precisely we use one .info file, that gives us information about the nodes, one or more .prec files that contain data about the predecessors of each node and one or more .succ files that, similarly, contain data about the successors of each node. We refer to all these file related to a single graph as the .ips multifile. We considered also the following graph file formats:

- **Text** In this file format we represent a graph as the list of its edges; the file is in plain text, and every edge is noted with the numbers of the nodes it connects.
- **Dimacs** This is the dimacs graph format; details can be find at
- **Edges** This file (or multifile) is simply a list of edges, where each edge is represented by the identifier of the source and ending node, written in binary.
- **Succ** In this file there there is first the identifier of a node, followed by the number of successors, followed by the identifiers of the successor nodes.

Figures 1 to 6 provide the representations of an example graph in all the formats described above.

4 List of procedures

In this section we detail the programs included in the library. We divide them in six categories: generators, measurers, search algorithms, bow-tie discovering, file converters and miscellaneous. All the procedures are implemented using external algorithms, except for two programs (bowtie, semiext_dfs) that are semiexternal. Indeed `bowtie` requires 2 bits for every node in the original graph; `semiext_dfs` uses 12 bytes and 1 bit for each node (For details about semiexternal dfs algorithms see [11]).

[1]This limit can be changed but we preferred, for portability reasons, to use a multifile format.

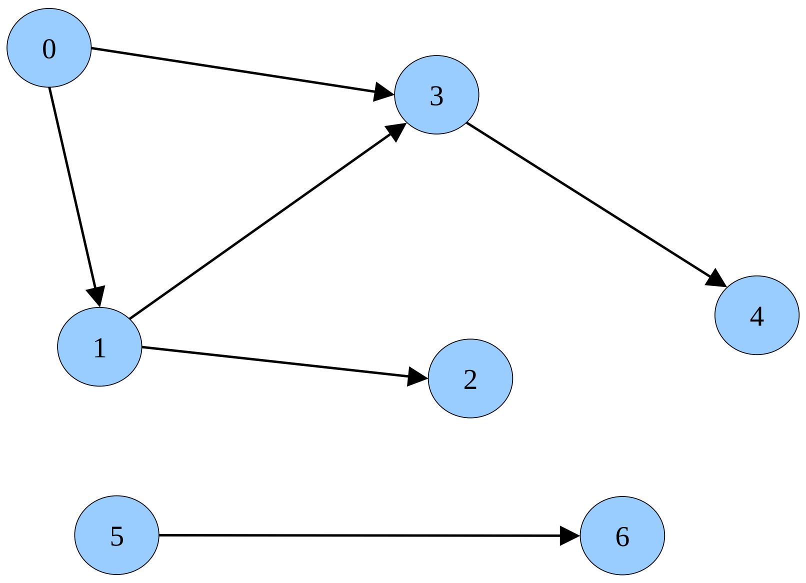

Figure 1: Example graph

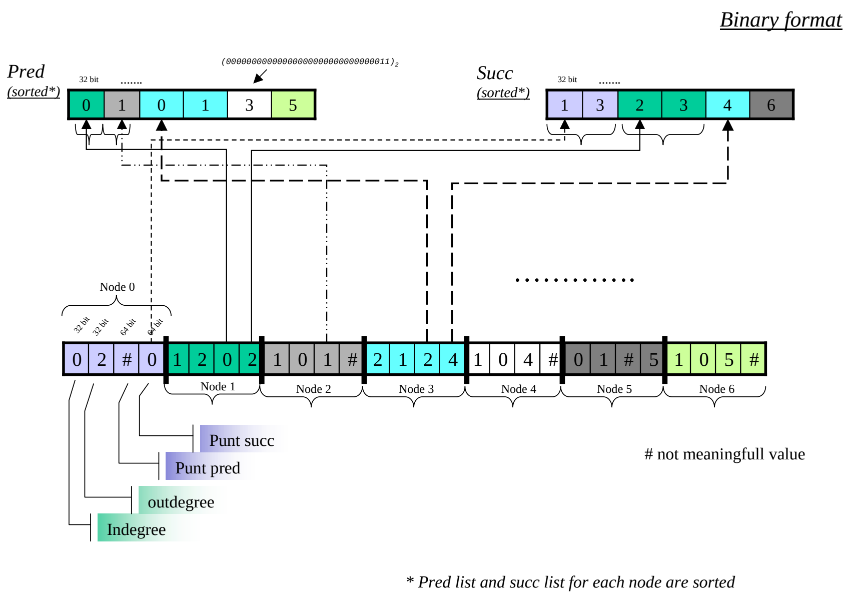

Figure 2: Ips format

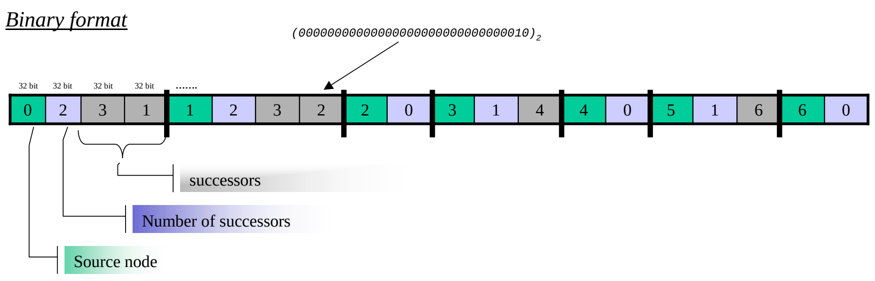

Figure 3: Succ format

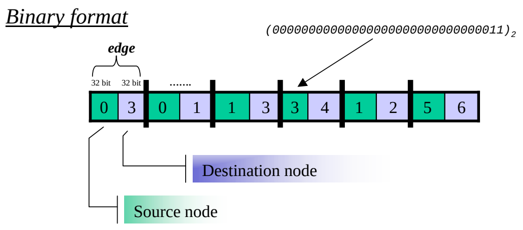

Figure 4: Edges format

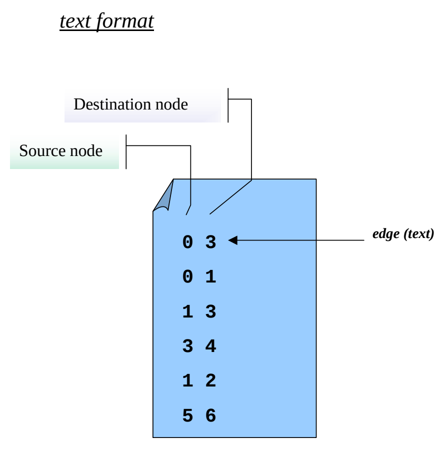

Figure 5: Text format

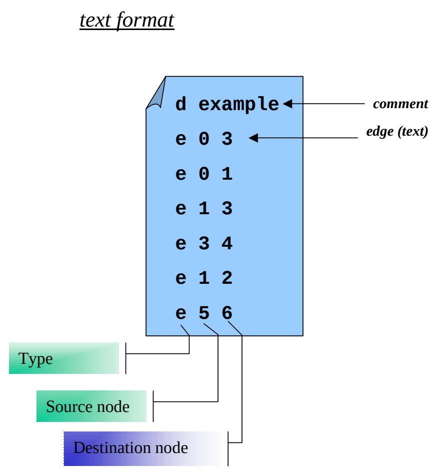

Figure 6: Dimacs format

4.1 Graph generators

Copying Model

This program generates a graph according to the copying model proposed by Kumar *et al.* [7]. The usage is as follows:

```
copymodel memory num_nodes numSucc alpha seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`alpha` copying parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Network Growth Model

This program generates a graph according to the Network Growth model proposed by Pennock *et al.* [10]. The usage is as follows:

```
netgrowthmodel memory num_nodes numSucc beta seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`alpha` uniform choice parameter (in the range 0-100)
`beta` uniform choice parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Evolving Network Model

This program generates a graph according to the Evolving Network model proposed by Albert *et al.* [1]. The usage is as follows:

```
evolvingmodel memory num_nodes numSucc seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Multi-Layer Model

This program generates a graph according to the Multi-Layer model proposed by Laura *et al.* [9]. We distinguish six different cases that depend on the random graph generator that is used to model every single layer. Therefore we have:

1. **Copying Model**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled according to the Copying Model of Kumar *et al.* [7]. The usage is as follows:

```
layermodel memory -LC|-LC_nd num_nodes numSucc numLayers layersPerEdge
alpha seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`alpha` copying parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Note that instead of the -LC switch, you can use the -LC_nd that allows nodes in a single layer not necessarily to be distinct ones

2. **Evolving Network Model**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled according to the Evolving Networks of Albert *et al.* [1] The usage is as follows:

```
layermodel memory -LB|-LB_nd num_nodes numSucc numLayers layersPerEdge
seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Note that instead of the -LB switch, you can use the -LB_nd that allows nodes in a single layer not necessarily to be distinct ones

3. **Mixed: Copying Model and Evolving Network Model**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled using a mixed strategy between the Copying Model of Kumar *et al.* [7] and the Evolving Networks of Albert *et al.* [1]. A node either copies the link from the vertex prototype p, as in the Copying Model, or points a node with the *preferential attachment*, as in the Evolving Network model. The usage is as follows:

```
layermodel memory -LCB|-LCB_nd num_nodes numSucc numLayers layersPerEdge
alpha seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`alpha` copying parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Note that instead of the -LCB switch, you can use the -LCB_nd that allows nodes in a single layer not necessarily to be distinct ones

4. **Network Growth Model**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled according to the Network Growth Model of Pennock *et al.* [10]. The usage is as follows:

```
layermodel memory -LN1|-LN1_nd num_nodes numSucc numLayers layersPerEdge
beta seed nameOutputMultifile%d
```

`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`beta` uniform choice parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)

Note that instead of the -LN1 switch, you can use the -LN1_nd that allows nodes in a single layer not necessarily to be distinct ones

5. **Network Growth Model with random parameter**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled according to the Copying Model of Kumar *et al.* [7]. The usage is as follows:

```
layermodel memory -LN2|-LN2_nd num_nodes numSucc numLayers layersPerEdge
betamin betamax seed nameOutputMultifile%d
```
`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`betamin` minimum value of the uniform choice parameter (in the range 0-100)
`betamax` maximum value of the uniform choice parameter (in the range 0-100)
`seed` seed for the random number generator
`output_file_name%` name of the output file (MUST include the string %)

Note that instead of the -LN2 switch, you can use the -LN2_nd that allows nodes in a single layer not necessarily to be distinct ones

6. **The model is randomly chosen amongst the ones described above**. This generates a graph according to the Multi-Layer Model in which the single layer is modelled according to the Copying Model of Kumar *et al.* [7]. The usage is as follows:

```
layermodel memory -RND|-RND_nd num_nodes numSucc numLayers layersPerEdge
alphaCopying alphaMixed beta seed nameOutputMultifile%d
```
`memory` memory available (MB)
`numnodes` number of graph's nodes
`numSucc` number of successor of every node (constant)
`numLayers` total number of layers
`layersPerEdge` number of layers per edge
`alphaCopying` copying parameter of the Copying model (in the range 0-100)
`alphaMixed` copying parameter of the Mixed model (in the range 0-100)
`beta` uniform choice parameter (in the range 0-100)
`seed` seed for the random number generator
`nameOutputMultifile%d` name of the output file (MUST include the string %d)

Note that instead of the -RND switch, you can use the -RND_nd that allows nodes in a single layer not necessarily to be distinct ones

4.2 Graph measurers

In this section we detail the routine that measure properties of an input graph, represented as always in our multifile .ips format.

Degree

This program computes the indegree and outdegree distribution of the input graph, in the IPS format. The indegree and outdegree distributions are stored in two files named as specified in the command line options, respectively with the suffices `.in` and `.out`. The usage is therefore as it follows:

```
degree memory nameInputMultifile%dfileNameTemp%d nameOutputMultifile%d
```
`memory` memory available (MB)
`nameInputMultifile%d` is the name of the input multifile (MUST include the string %d)
`fileNameTemp%d` is the name of a temporary multifile (MUST include the string %d)
`nameOutputMultifile%d` the name of the output file
(MUST include the string %d)

pagerank

This routine computes the PageRank on the input graph. The output are two file that contain, respectively, the distribution of pagerank values and, for each node, its pagerank value. The residual value is the difference of the absolut value of two consecutive iterations of the algorithm. The command line usage is as follows:

```
pagerank memory nameInputMultifile%dc residual maxIter fileTxtOutDistr%d
fileTxtOutNodes%d colums
```

`memory` memory available (MB)
`nameInputMultifile%d` name of the input multifile
`residualT` threshold value of residual; if residual is less then this value, then the program halts
`c` PageRank parameter: probability to remain in the page (the value is in the range 0 up to 1; suggested is 0.85)
`maxIter` maximum number of iterations allowed
`fileTxtOutDistr%d` name of the output (text) multifile where the distribution of pagerank values are stored (MUST include the string %d)
`fileTxtOutNodes%d` name of the output (text) multifile where pagerank values are stored (MUST include the string %d)
`colums` specify the info to write in fileTxtOutNodes%; possible values are N,I,O,P,- where

- N: column with the number of the node
- I: column with the indegree value of the node
- O: column with the outdegree value of the node
- P: column with the pagerank value of the node
- -: nothing is written

correlation

This routine computes the degree correlation functions of the input graph. We can define eight distinct of such quantities that give us the correlation between the in-degree (out-degree) of the followers (predecessors) as a function of the in-degree (out-degree) distribution; we have all the combination $\{I|O\}\{f|p\}\{I|O\}$ where the first letter ($\{I|O\}$) means that we are plotting the average in-degree (or out-degree), the second($\{f|p\}$) that we are looking at the followers (or predecessors), and the last one ($\{I|O\}$) that we are drawing the plot as a function of the in-degree (or the out-degree). The usage is as follows:

```
correlation memory neighbors degree degNeighbors nameInputMultifile%d
nameTemporaryMultifile%d nameOutputTextMultifile%d
```

`memory` memory available (MB)
`neighbors -pred|- succ` specify that the analysis is based on the predecessors or successors
`degree -indeg|-outdeg|-node` if we want the results collected by the indegree, the outdegree or for each node
`degNeighbors -indeg|-outdeg` if we are interested in the indegree or outdegree of the predecessors or successors (as specified in `neighbors`)
`nameInputMultifile%d` name of the multifile of the input graph (MUST include the string %d)
`nameTemporaryMultifile%d` name of a temporary multifile (MUST include the string %d)
`nameOutputTextMultifile%d` name of the output file, that is a text multifile (MUST include the string %d)

clique

This routine measure the number of clique, i.e. the number of bipartite cores, of the input graph. The command line options are as follows:

```
clique memory nameInputMultifile%d nameTemporaryMultifile%d
nameOutputFile {i,j}+
```

`memory` memory available (MB)
`nameInputMultifile%d` name of the multifile of the input graph (MUST include the string %d)
`nameTemporaryMultifile%d` name of a temporary multifile (MUST include the string %d)
`nameOutputFile` name of the output file
$\{i, j\}^+$ (one or more) type of bipartite cores to look for; examples are: $3, 5$ or $3, 5$ $3, 6$ or $3, 5$ $2, 6$ $3, 8$.

4.3 Search algorithms

bfs

This utility performs the Breath First Search of a graph. The command line options are as follows:

```
bfs memory nameInputMultifile%d nameTempMultifile%d
nameOutputMultifile%d startingNode directionOption
```

`memory` memory available (MB)
`nameInputMultifile%d` name of the multifile of the input graph (MUST include the string %d)
`nameTempMultifile%d` name of a temporary multifile (MUST include the string %d)
`nameOutputMultifile%d` name of the output multifile (MUST include the string %d)(*)
`startingNode` starting node of the bfs
`directionOption` -FORWARD — -BACKWARD — -BOTH(**) to specify the direction of visit

(*) The basic multiFile contains all the visited nodes, grouped by level. (Different levels are separated by -1) The second multiFile with extension '.report' contains report informations (**)if the direction option is BOTH, the edges are considered undirected.

semiext_dfs

Utility that performs a Depth Firsts Search of a graph using a semiexternal algorithm. For each node at least (12 + 1/8) bytes are required. The command line options are as follows:

```
semiext_dfs memory inputGraphMultiFile%d outputSccMultifile%d
[Options]...
```

`memory`available memory in MB
`inputGraphMultiFile%d` name of the multifile of the input graph (MUST include the string %d)
`outputSccMultifile%d` output dfs forest in adjacency list format (MUST include the string %d)
Options:

- `-b, -backward` to execute a backward visit
- `-f, -output_f` nodes are listed in the order they are found the first time instead of the order they finish their DFS calls
- `-force` executes the visit even if the main memory is not enough
- `-n, -sourceNode=<N>` performs a DFS starting from node <N>
- `-o, -file_ord=<fileOrd%d>` visit the DFS forest following the nodes order specified in <fileOrd%d>
- `r, -roundsMax=<num>` numbers of rounds after which the utility stops
- `-t, -iterationsMax=<num>` max number of iterations to do
- `-m, -minutesMax=<minutes>` max elapsed time after which the utility stops

4.4 Bow-tie discovering

bowtie

This program searches for the Bow-Tie regions. It uses a semi-external algorithm that requires at least 2 bits for each graph node. The command line options are as follows:

```
bowtie [Options]...  memory graphFile%d fileSetSCC%d
```

`memory` available memory in MB
`graphFile%d` name of the multifile of the input graph (MUST include the string %d)
`fileSetSCC%d` output multiFile name; it contains the nodes in the CORE (as adjacency list)(MUST include the string %d)
Options:

- `-force` performs the visit even if the main memory is not enough (slower: it uses swap memory)
- `-r, -random[=<NumIterations>]` searches CORE using <NumIterations> random nodes
- `-f, -file=<nodesFile%d>` searches CORE using all the nodes stored in <nodesFile%d>
- `-s, -seed=<semeRandom>` specifies the seed of the random generator (it must be used with -random option)
- `-I, -IN=<fileSetIN%d>` computes all the nodes that reach the CORE
- `-O, -OUT=<fileSetOUT%d>` computes all the nodes that are reached from the CORE
- `-T, -TENDRILS=<fileSetTENDRILS%d>` computes TENDRILS and TUBES regions
- `-D, -DISC=<fileSetDISC%d>` computes DISCONNECTED region
- `-b, -bits` saves nodes of each region (IN,OUT,TENDRILS,DISCONNECTED) as bit vector and not as a list of nodes (default)
- `-a, -all` saves all the computed SCCs in <fileSetSCC%d> and not only the bigger one
- `-p, -percentMax=<perc>` minimum dimension of acceptable CORE (percent of nodes)
- `-n, -numSCCMax=<num>` maximum number of SCCs to consider
- `-m, -minutesMax=<minutes>` time limit of computation

scc.script

This script performs Bow-Tie measurements and calculates SCCs in the graph. It uses ./temp_scc directory to store additional files used in the process. The command line options are as follows:

```
./scc.script memory IPS_graph%d output_SCC%d
```

`memory` available RAM in MB
`input_graph%d` input graph multifile name (MUST include the string %d)
`output_SCC%d` output SCC in adjacency list format (MUST include the string %d)

This utility creates five multifiles that contain nodes of each bow-tie region. (INset%d, OUTset%d, DISCset%d, TENDRILSset, TENDRILSset.tubes, DISCset%d)

4.5 Format converters

As we have seen in section 3 we considered five different file formats:

ips Our multifile format.

succ For every node, the number of successors followed by the successors.

edges A list of edges in binary file (32 bit each node).

text A list of edges in a text file.

dimacs The Dimacs graph format [**?**].

We provide eight script that are able to convert our .ips multifile from/to any other file type.

Ips2xxx converters:

```
ips2text memory -deleteSource|-saveSource nameInputMultifile%d
nameOutputFile
ips2succ memory -deleteSource|-saveSource nameInputMultifile%d
nameOutputFile
ips2edges memory -deleteSource|-saveSource nameInputMultifile%d
nameOutputFile
ips2dimacs -deleteSource|-saveSource memory nameInputMultifile%d
nameOutputFile
```

`memory` memory available (MB)
`-deleteSource|-saveSource` specify whether to delete or keep the input file
`nameInputMultifile%d` name of the multifile of the input graph (MUST include the string %d)
`nameOutputFile` FULL name (including the extension) of the of the output file

Xxx2ips converters:

```
text2ips memory -deleteSource|-saveSource nameInputfile
nameOutputMultifile
succ2ips memory -deleteSource|-saveSource nameInputMultifile
nameOutputMultifile%d
edges2ips memory -deleteSource|-saveSource nameInputMultifile
nameOutputMultifile%d
dimacs2ips memory -deleteSource|-saveSource nameInputfile
nameOutputMultifile%d
```

`memory` memory available (MB)
`-deleteSource|-saveSource` specify whether to delete or keep the input file
`nameInputFile` name of the file of the input graph
`nameInputMultiFile%d` name of the multifile of the input graph (MUST include the string %d)
`nameOutputMultiFile%d` name of the output multifile (MUST include the string %d)

4.6 Miscellaneous

subgraphrand

This routine, given a multifile (in the .ips format) in input, representing a graph $G = (V, E)$, creates the subgraph G' induced by a random selection of the nodes. More formally, a random subset $V' \subseteq V$ is chosen; if $v_1, v_2 \in V'$ and the edge $e_{1,2} = (v_1, v_2)$ is in E, the edge $e_{1,2}$ is E'. It follows the command line usage:

```
subgraphrand memory -RU|-RI maxIterations nameInputMultifile%d
nameOutputMultifile%dnumNodesInSubgraph seed
```

`memory` memory available (MB)
`-RU|-RI` specify whether to select the nodes of the subgraph with a uniform (-RU) or indegree based (-RI) distribution probability
`maxIterations` specify the maximum number of iterations to randomly extract the set V'; we suggest not to exceed the value 10
`nameInputMultifile%d` name of the multifile of the input graph (.ips format) (MUST include the string %d)
`nameOutputMultifile%d` name of the multifile of the output graph (.edges format) (MUST include the string %d)
`numNodesInSubgraph` the (expected) number of nodes of the subgraph
`seed` the seed for the random number generator

rewiring

This routine allow to modify an input graph by i) inverting the direction of randomly selected edges, or ii) changing the successor in randomly selected edges, or iii) by adding new edges to the graph. The usage is as follows:

1. Inverting the direction of randomly selected edges. The usage is as follows:

```
rewiring -IE -deleteSource|-saveSource nameInputMultifile%d
probInversion seed nameOutputMultifile%d
```

`-deleteSource|-saveSource` specify whether to delete or keep the input file
`nameInputMultifile%d` name of the input multifile (.edges format) (MUST include the string %d)
`nameOutputFile%d` name of the output multifile (.edges format) (MUST include the string %d)
`probInversion` probability (in the range $0 - 100$) of inverting an edge
`seed`

2. Changing the successors in randomly selected edges. Command line options are the following:

```
rewiring -CE -deleteSource|-saveSource nameInputMultifile%d
probChangingSucc seed nameOutputMultifile%d
```

`-deleteSource|-saveSource` specify whether to delete or keep the input file
`nameInputMultifile%d` name of the input multifile (.edges format) (MUST include the string %d)
`nameOutputMultifile%d` name of the output multifile (.edges format) (MUST include the string %d)
`probChangingSucc` probability (in the range $0 - 100$) of changing the successor
`seed` the seed for the random number generator

3. Adding new edges. It follows the command line usage:

```
rewiring -AE -deleteSource|-saveSource nameInputMultifile%d m seed
nameOutputMultifile%d
```

`-deleteSource|-saveSource` specify whether to delete or keep the input file
`nameInputMultifile%d` name of the input multifile (.edges format) (MUST include the string %d)
`nameOutputMultifile%d` name of the output multifile (.edges format) (MUST include the string %d)
`m` number of edges to be added
`seed` the seed for the random number generator

5 Example

In this section we present an example of usage of our library.

We want to generate a graph based on the evolving network model, with 10000 nodes. We also want to compute in/outdegree distribution, cliques 2,3 and Pagerank values distribution on the generated graph.

We assume that the current directory contain the binary files of the library and the computer has 100MB RAM.

The commands are the following:

1. With the `evolvingmodel` routine, we generate a graph in edges format with 10000 nodes and 7 successors per node. We choose 1234567 as the value for the seed and `edges.%d` as the name of the output multifile.
   ```
   ./evolvingmodel 100 10000 7 1234567 edges.%d
   ```
2. We perform the conversion from the .edges format multifile to the .ips format with the script edges2ips (we recall that the graph measurers works with input file in the .ips format)
   ```
   ./edges2ips 100 edges.%d graph.%d
   ```
3. We use the `degree` routine to compute the indegree/outdegree distribution (using -distr option to calculate the distribution); the output file name is distr_degree.%d.txt, a text multifile.
   ```
   ./degree 100 -distr graph.%d temp.%d distr_degree.%d.txt
   ```
4. We compute the clique (2,3) with the following command
   ```
   ./cliques 100 graph.%d temp.%d 2,3
   ```
5. Last, we compute the pagerank distribution using the `pagerank` utility (with parameters: c=0.85, residual=0.00002, max_iterations=50).
   ```
   ./pagerank 100 graph.%d 0.85 0.00002 50 distr.%d.txt nodes.%d.txt -
   ```

References

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

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

[3] G. Caldarelli, P. De Los Rios, L. Laura, S. Leonardi, and S. Millozzi. A study of stochastic models of the webgraph. Technical Report 04-03, DIS - University of Rome La Sapienza, 2003.

[4] D. Donato, L. Laura, S. Leonardi, and S. Millozzi. Large scale properties of the webgraph. *European Journal of Physics B*, 2004. To appear.

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

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

[7] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random graph models for the web graph. In *Proc. of 41st FOCS*, pages 57–65, 2000.

[8] 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.

[9] 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.

[10] 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.

[11] 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.