The Connectivity Server: fast access to linkage information on the Web

Krishna Bharat[a], Andrei Broder[a], Monika Henzinger[a],
Puneet Kumar[a], and Suresh Venkatasubramanian[b]
[a]*DIGITAL, Systems Research Center,*
130 Lytton Avenue, Palo Alto, CA 94301, U.S.A.
bharat@pa.dec.com, broder@pa.dec.com,
monika@pa.dec.com, and pkumar@pa.dec.com
[b]*Computer Science Department, Stanford University,*
Palo Alto, CA 94301, U.S.A.
suresh@cs.stanford.edu

Abstract

We have built a server that provides linkage information for all pages indexed by the AltaVista search engine. In its basic operation, the server accepts a query consisting of a set L of one or more URLs and returns a list of all pages that point to pages in L (predecessors) and a list of all pages that are pointed to from pages in L (successors). More generally the server can produce the entire neighbourhood (in the graph theory sense) of L up to a given distance and can include information about all links that exist among pages in the neighbourhood. Although some of this information can be retrieved directly from Alta Vista or other search engines, these engines are not optimized for this purpose and the process of constructing the neighbourhood of a given set of pages is slow and laborious. In contrast our prototype server needs less than 0.1 ms per result URL. So far we have built two applications that use the Connectivity Server: a direct interface that permits fast navigation of the Web via the predecessor/successor relation, and a visualization tool for the neighbourhood of a given set of pages. We envisage numerous other applications such as ranking, visualization, and classification.

Keywords

Connectivity; Web links; Web graph; Web visualization; Search

1. Introduction

Information about explicit links between Web pages or the topology of the Web has been often identified as a valuable resource for data mining. (See for example [5], [6] and references therein.) However previous research and commercial efforts that tried to use this information in practice have collected linkage data either within a narrow scope or in an inefficient and ad-hoc manner.

In an attempt to alleviate this problem we have built a server, called the Connectivity Server, that provides linkage information for all pages indexed by the AltaVista search engine. In its basic operation, the server accepts a query consisting of a set *L* of one or more URLs and returns a list of all pages that point to pages in *L* (predecessors) and a list of all pages that are pointed to from pages in *L* (successors). More generally the server can produce the entire neighbourhood (in the graph theory sense) of *L* up to a given distance and can include information about all the links that exist among pages in the neighbourhood.

Although some of this information can be retrieved directly from Alta Vista or other search engines, the search engines are not optimized for this purpose and the process of constructing the neighbourhood of a given set of pages is slow and laborious. In contrast our prototype server needs less than 0.1 ms per result URL.

To test our prototype we have built two applications, described further below. One is a simple direct interface that allows a user to navigate the Web by determining quickly all predecessors and successors of a given page. The other is a visualization tool for the neighbourhood of a given set of pages.
Using a server to provide linkage information to interested clients has many more potential applications. Some examples of current research projects or commercial software that use linkage information are:

- Visualization tools
  - Microsoft's Site Mapping tool. (Used to be NetCarta.)
  - MAPA from Dynamic Diagrams.
  - IBM's Mapuccino, formerly WebCutter, described in [4].
  - Merzscope from Merzcom.
  - CLEARweb.
  - WebAnalyzer from InContext.
- Search tools
  - WebSQL described in [1].
  - The hyper search engine described in [5].
  - WebQuery described in [2].
- Ranking tools
  - Kleinberg's method for finding "Authoritative Sources In A Hyperlinked Environment" [3], discussed further below.
  - The Rankdex, a "Hyperlink search engine".
  - The PageRank system developed at Stanford, now part of the BackRub.

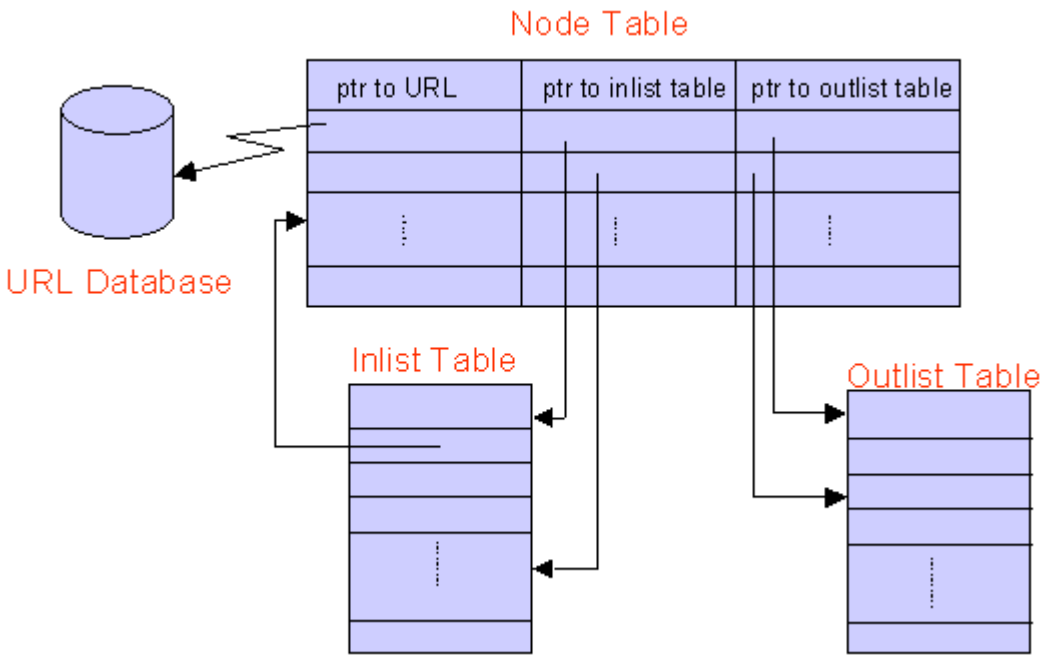
In a different direction, the Connectivity Server can be used to optimize the design and implementation of Web crawlers by offering data about the linkage graph such as statistics on the in and out degrees, good partitions of the graph for parallel crawling, etc.

2. Internal organization

2.1. Initial data structures

Representing a small graph is trivial. Representing a graph with 100 millions nodes and close to a billion edges is an engineering challenge.
We represent the Web as a graph consisting of nodes and directed edges. Each node represents a page and a directed edge from node *A* to node *B* means that page *A* contains a link to page *B*.
The set of nodes is stored in an array, each element of the array representing a node. The array index of a node element is the node's *ID*. We represent the set of edges emanating from a node as an adjacency list, that is for each node we maintain a list of its successors. In addition, for each node we also maintain an inverted adjacency list, that is a list of nodes from which this node is directly accessible, namely its predecessors. Therefore a directed edge from node *A* to node B appears twice in our graph representation, in the adjacency list of *A* and the inverted adjacency list of B. This redundancy in representing edges simplifies both forward and backward traversal of edges. To minimize fragmentation, elements of all adjacency lists are stored together in one array called the `Outlist`. Similarly elements of all inverted adjacency lists are stored in another array called the `Inlist`. The adjacency and inverted adjacency lists stored in each node are represented as offsets into the `Outlist` and `Inlist` arrays respectively. The end of the adjacency list for a node is marked by an entry whose high order bit is set (see Fig. 1) Thus we can determine the predecessors and the successors of any node very quickly.

Fig. 1. Representation of the graph.



A node in the Web-graph has an attached URL. Since URLs are rather long (about 80 bytes on average), storing the full URL within every node in the graph would be quite wasteful. (The storage requirement of a naive implementation would be about 8 gigabytes for 100 million URLs!) Instead the server maintains data structures that represents the *ID* to URL and URL to *ID* mappings.

After a full crawl of the Web, all the URLs that are to be represented in the server are sorted lexicographically. The index of a URL in this sorted list is its initial *ID* (see the discussion of updates below). Then the list of sorted URLs is stored as a delta-encoded text file, that is, each entry is stored as the difference (delta) between the current and previous URL. Since the common prefix between two URLs from the same server is often quite long, this scheme reduces the storage requirements significantly. With the 100 million URLs in our prototype we have seen a 70% reduction in size (see Fig. 2)

This reduction in storage requirements comes at a price, namely the slowdown of the translation. In order to convert a delta encoded entry back to its complete URL, one needs to start at the first URL and apply all the deltas in sequence until arriving at the URL in question. We avoid this problem by periodically storing the entire URL instead of the delta encoding. This entry is called a *checkpoint* URL. Therefore to translate a delta encoded URL, we need to apply the deltas starting from the last checkpoint URL rather than the first URL. The cost of the translation can be reduced by increasing the checkpoint frequency (see Fig. 3) To translate a URL to an internal ID we first search the sorted list of checkpoint URLs to find the closest checkpoint. Then the delta encoded list is searched linearly from that checkpoint URL until the relevant URL is reached. To speed up the the reverse translation from internal ID to an URL, the relevant node points directly to the closest checkpoint URL. As before the URL is computed by searching linearly from the checkpoint URL.

Fig. 2. Delta encoding the URLs.

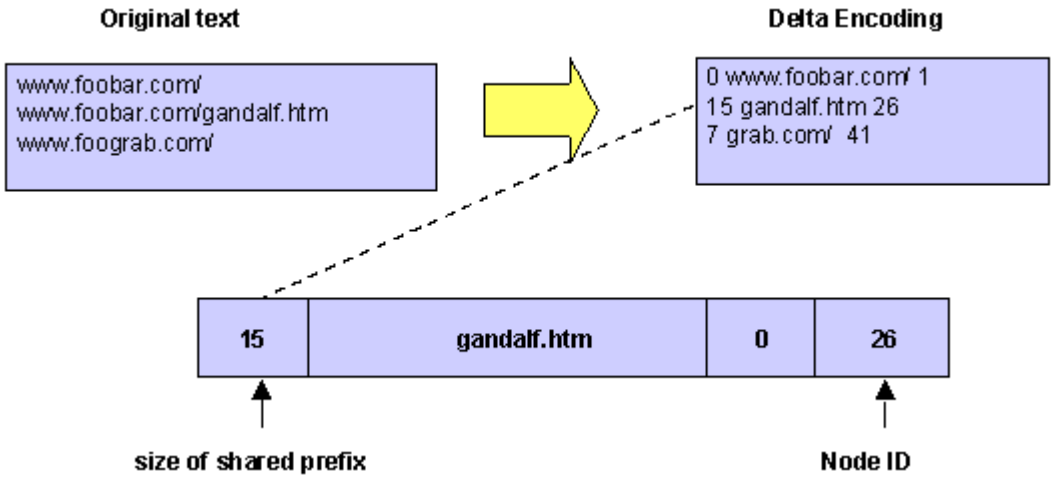
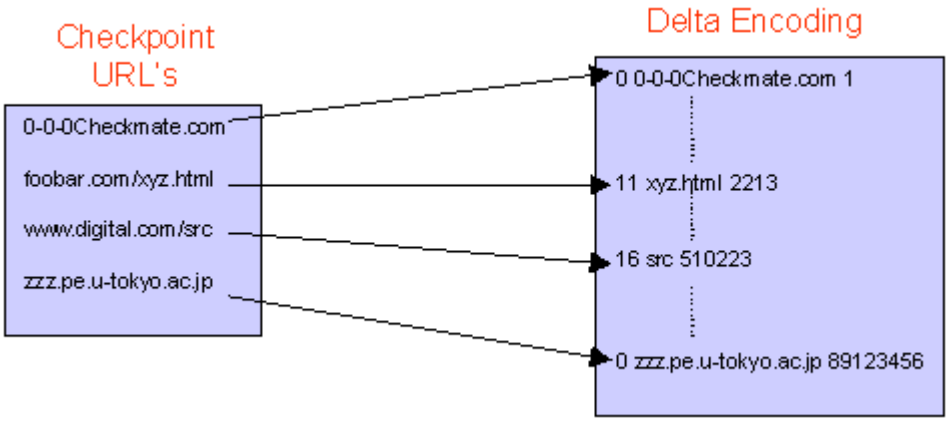


Fig. 3. Indexing the delta encoding.



2.2. Updates

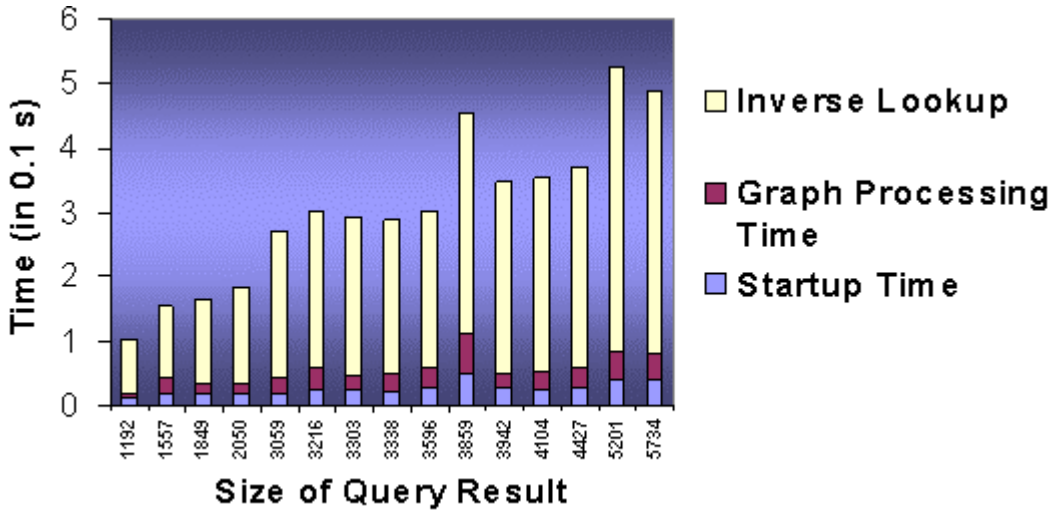
Since our structure is very tight, updates are not simple. Currently our design is to batch all the updates for a day. We view all the updates as a collection of nodes and edges to be added or deleted. All deletions can be done by marking the deleted edges and nodes in a straightforward manner. This requires an extra bit per edge and node. Additions are done as follows.

To allow for additions, we allocate initially larger tables than immediately necessary. For newly added nodes, we maintain a separate structure for the URL to id translation, organized as a string search tree. This tree contains all the newly added nodes and their assigned ID's in the main data structure. To update the `Outlist` table, the list of new edges is grouped by source. If the new `Outlist` associated to a node is longer than the old `Outlist`, space is allocated at the end of the current `Outlist` table. The update of the `Inlist` table is done similarly, except that edges are sorted by destination. Eventually the wasted gaps in tables consume too much space, and/or the additional node tree becomes too large and then the entire structure is rebuilt.

3. Performance

The Connectivity Server performs three steps to process queries: translate the URLs in the query to node IDs, explore the Web graph around these nodes and translate the IDs in the result set back to URLs. Thus the time needed to process queries is proportional to the size of the result set. On a 300 MHz Digital Alpha with 4 GB memory, the processing time is approximately 0.1 ms/URL in the result set. Figure 4 shows the timings for 15 different queries where the answer size varies from 1192 to 5734 URLs. As shown in Fig. 4 the third step takes up most of the processing time, i.e. 80%. The remainder time is shared equally between steps one and two. Therefore, applications that can work with internal IDs can expect an even faster processing time of about 0.01 ms/URL.

Fig. 4. Query processing times.



4. Applications

4.1. Direct interface

The direct interface provides basic query access to the Connectivity Server. It provides two kinds of query interfaces: a simple query interface and an advanced query interface.
The simple query interface (shown in Fig. 5) allows the user to type in a URL and provides three buttons: one to retrieve the predecessors (left arrow), one to retrieve the successors (right arrow), and one to retrieve both sets of neighbours (double arrow).

Fig. 5. The Connectivity Server simple query interface.

Figure 6 shows a subset of the results of the simple query from Fig. 5. This lists the original URL and the requested URLs. Each URL is displayed with a hyperlink to access the corresponding page, and left and right arrow buttons to access its predecessors and successors respectively.

Fig. 6. Results of a simple query.

The advanced query interface (see Fig. 7) gives the user more options. The user can specify:

- The radius (i.e. distance to the given URL) of the neighbourhood that the user wants to retrieve.
- Limits on the number of incoming and outgoing edges that are explored from any node.
- A Display mode which gives three options:
  - A hierarchical representation called the *tree view* (see Fig. 8 below). This shows the original URL without indentation, the requested URLs at distance 1, indented once, the requested URLs at distance 2, indented twice and grouped under the corresponding distance-1 URL, etc.
  - A list representation of all URLs at exactly the specified distance.
  - A list representation of all URLs up to the specified distance, sorted by distance.

As before, URLs in the result set are displayed with navigation icons to access their neighbours.

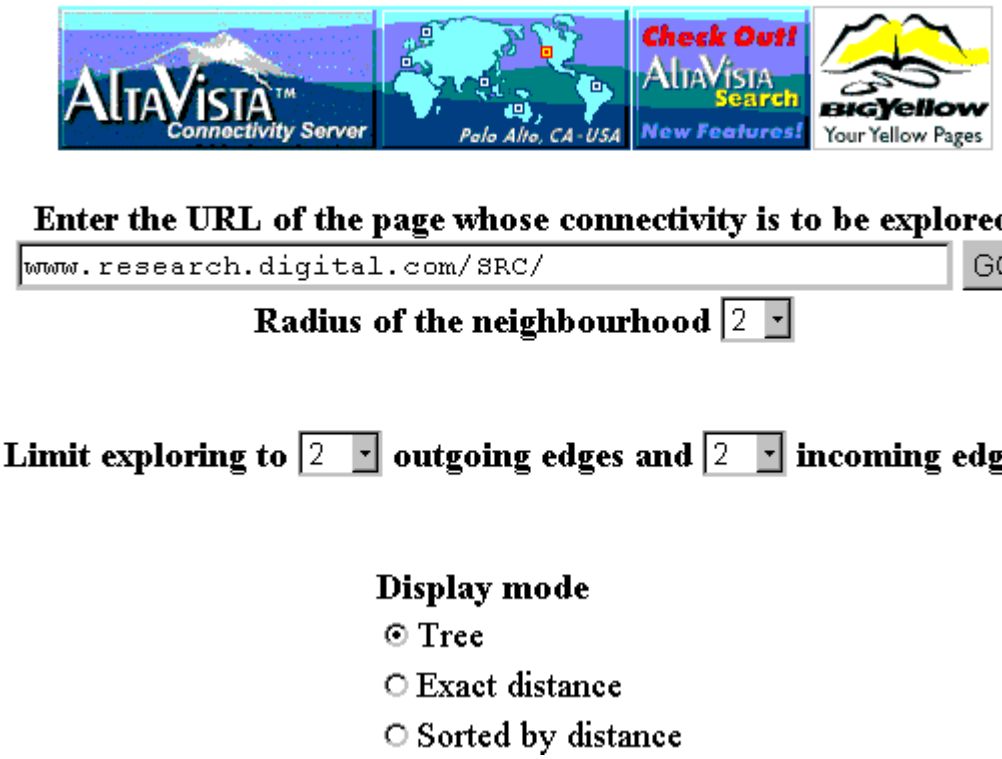


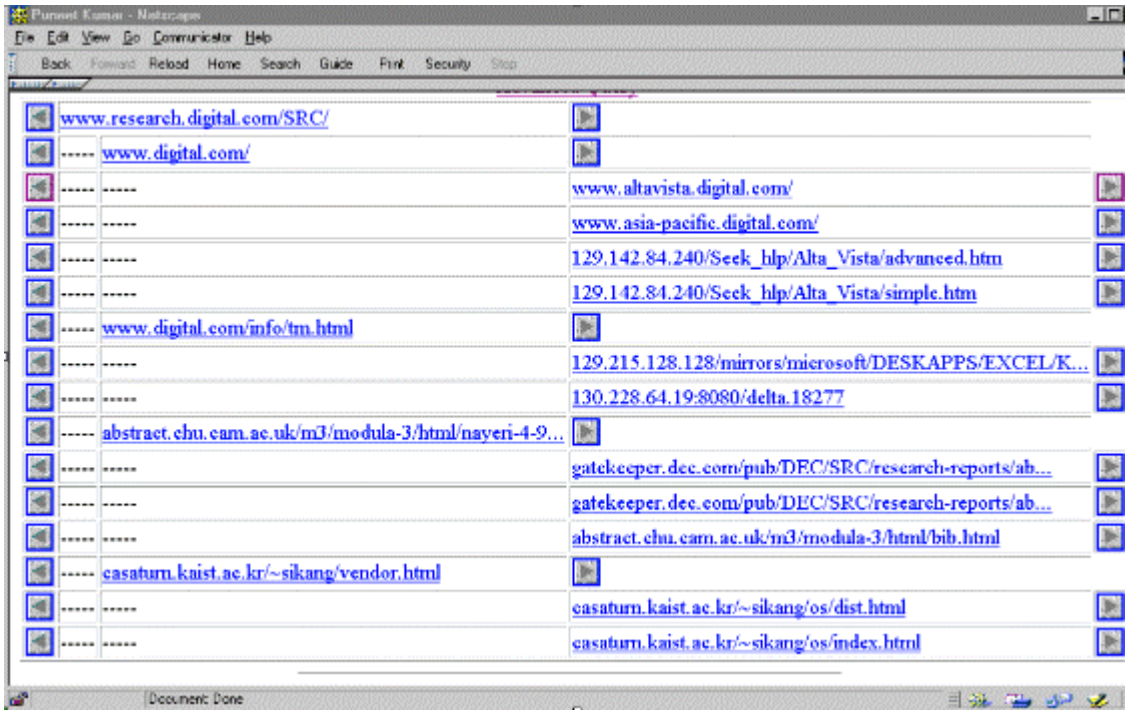
Fig. 7. The Connectivity Server advanced query interface.

E r r e u r! Si g

net ondéfini.

Figure 8 shows a tree-view display for the advanced query in Fig. 7. Two incoming and two outgoing edges are considered for each URL to compute a neighbourhood of radius two. For example, `www.digital.com` *is a distance-1 neighbour of* `www.research.digital.com/SRC/` *and* `www.altavista.digital.com` *is a distance-1 neighbour of* `www.digital.com`*.*

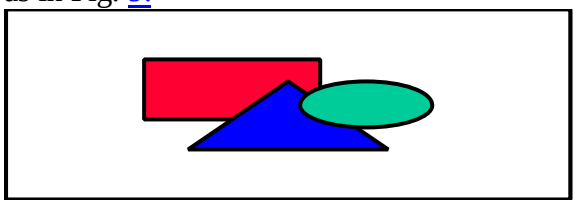
Fig. 8. Results of an advanced query in tree mode.



4.2. Visualization of connectivity data

The second application is more complex and makes use of the fact that the Connectivity Server can compute the whole neighbourhood of a set of URLs in the graph theoretic sense. We will define this more precisely: A *Neighbourhood Graph* is the graph induced by a set *L* of start pages and their distance-1 neighbourhood. (That is, *L*, all the predecessors of *L*, all the successors of *L*, and all the edges among them.) Kleinberg [3] showed that if *L* is the set of pages returned by a search engine for a specific query then the neighbourhood graph of *L* can be analyzed to detect useful pages and to rank the pages (see below). Our application is a tool to visualize this neighbourhood graph and to run ranking computations on it. We call the initial set *L* of pages the *Start Set*. The set of all pages that are predecessors of the Start Set and do not belong to the Start Set is called the *Back Set*. In practice we consider at most 50 such back pages per Start Set node, since the in-degree of a node on the Web can be very large. Similarly pages that are successors of the Start Set and belong to neither of the other sets constitute the *Forward*

Set. Our visualization system computes the graph induced by these three sets and lays them out as in Fig. 9.



Erreur! Signet non défini.

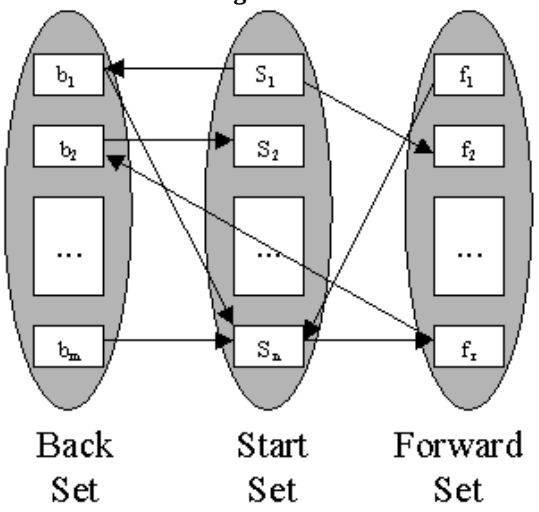


Fig. 9. The sets comprising the Neighbourhood Graph and their layout.

Erreur! Signet non défini.

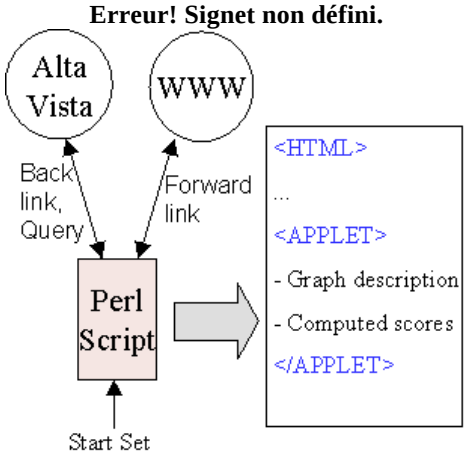


Fig. 10. The old setup for building the Neighbourhood Graph.

4.2.1. Computing the Neighbourhood Graph

Before the Connectivity Server existed we used a Perl script to compute the Neighbourhood Graph using AltaVista and direct access to the World Wide Web (see Fig. 10). For each page in the Start Set we used AltaVista `link:` queries to determine incoming links, which we call *Back Links*. These define the Back Set. Additionally we fetched each page in the Start Set to determine its *Forward Links*, which defined the Forward Set. All Forward and Back Set pages were fetched to compute the connectivity between them. We avoided using AltaVista `link:` queries for this since we wanted to have the most up to date information.

We make some modifications to the Neighbourhood Graph for our application. Edges between pages on the same host are first removed from the graph. Any nodes in the resulting graph that have no incoming or outgoing links are removed as well.

After the above filtering, for Start Sets of size 200 and with in-degree restricted to 50, the average Back Set size was about 180 and the average Forward Set size was about 970. It took on the order of two hours to construct the graph.

In our new implementation, we use the Connectivity Server as follows: The application gives the Start Set URLs to the Connectivity Server, which returns an adjacency list for the unfiltered Neighbourhood Graph. This takes under a minute. Since filtering takes in the order of a few seconds, the second implementation is two orders of magnitude faster.

4.2.2. Connectivity analysis

Our interest in the Neighbourhood Graph was motivated by the possibility of analyzing the graph's connectivity to find "useful" pages. Specifically, we were interested in implementing Kleinberg's algorithm for finding *authority* pages and good directories (called *hubs*) for a given user query [3].

A user query is translated into an AltaVista query. The top 200 results from AltaVista form the Start Set, from which the Neighbourhood Graph is built, as described above. There are two

scores associated with each page: a *hub score*, H, and an *authority score*, A. Initially all scores are set to 1. Then the following steps are iteratively executed until the scores converge:

1. For all nodes i, H[i] = Sum over all j, s.t. (i,j) is an edge {A[j]}
2. For all nodes j, A[j] = Sum over all i, s.t. (i,j) is an edge {H[i]}
3. Normalize H[]
4. Normalize A[]

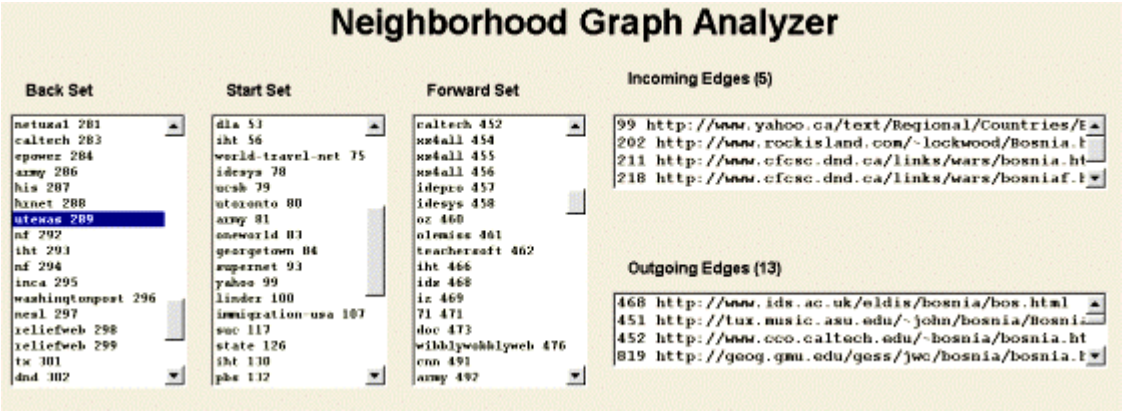
As shown by Kleinberg, the nodes with large A values are likely to be good authority pages, and the nodes with large H values are likely to be good hubs. The computation converges to a stable ranking of scores in about 10 iterations. We compute 150 iterations which takes about 2 minutes.

4.2.3. Views of the Neighbourhood Graph

Our visualization consists of several interactive views which are synchronized. Selecting a page in one view causes all the other views to update to the selected page as well. The figures below show a visualization of the neighbourhood graph for the result set of the AltaVista query: `bosnia`.

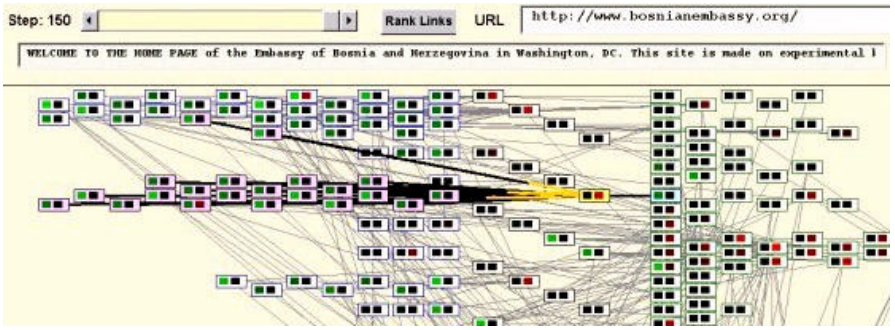
The first view is the *List View* that lists the elements of the three sets textually and shows incoming and outgoing edges of the current selection (see Fig. 11). For brevity URL names are sometimes abbreviated to shorter labels computed by concatenating the most significant part of the host name and a unique identifier (e.g., "utexas 289"). The same naming scheme is used to identify pages in other views as well whenever space is limited.

Fig. 11. The List View.



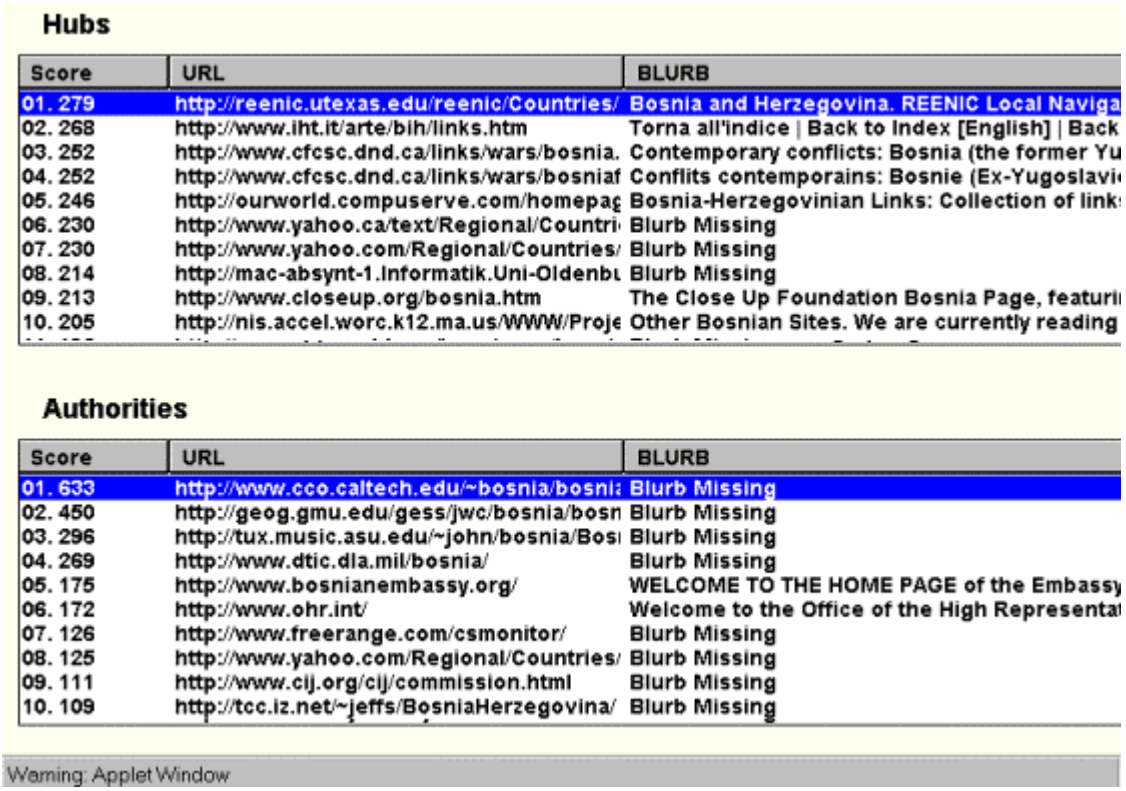
The Graph View (see Fig. 12) displays the connectivity between pages graphically. Each page is represented by a rectangular icon, and edges are drawn as straight lines between pairs of page icons. The direction of the edge is usually not displayed unless one of the nodes is selected. If a node is selected, all incident edges are highlighted. An edge is highlighted in black at the source node and in yellow at the sink node. With each node's icon its computed attributes are represented by a pair of red and green colour-coded boxes. The larger the attribute the greater is the intensity of the box's colour. In the Kleinberg application, we use the red box represents the page's authority score, and the green box represents the hub score.

Fig. 12. The Graph View.



At the top of the window we have a *Step* control to view the state of the computation at various iterations. For any given iteration, the "Rank Links" button brings up the *Ranked Pages View* which shows rankings of pages based on various attributes. Finally, for a selected page the URL and the first few words on the page (available if the page was actually fetched) are displayed.
The *Ranked Pages View* (see Fig. 13) displays various rankings of pages based on the ranking analysis algorithm being used. Currently we rank the nodes according to the authority and hub scores computed by Kleinberg's algorithm. Hence, there are two ranked lists. Double-clicking on a listing in the Ranked Pages View causes the corresponding page to be loaded into a browser window.

Fig. 13. The Ranked Pages View.



For example, the best hub and authority by the ranking in the above figure are "http://reenic.utexas.edu/reenic/Countries/Bosnia_and_Herzegovina/bosnia_and_herzegovina.html" and "http://www.cco.caltech.edu/~bosnia/bosnia.html", respectively. In this case the best hub and the best authority appear to be considerably more relevant than the places highly ranked by various search engines that we tried with the same query.

5. Conclusions

Our server shows that it is possible to provide linkage information for a significant portion of the Web (all pages indexed by AltaVista) at a fairly high speed. The applications that we have built

exemplify its use for navigation, visualization, and ranking, but we view this server as an enabling technology that can be the basis of many other applications, some maybe yet to be discovered.

References

[1]

G.O. Arocena, A.O. Mendelzon, and G.A. Mihaila,
Applications of a Web query language,
in: *Proc. of the 6th International World Wide Web Conference*, 1997, pp. 587–595,
http://www6.nttlabs.com/papers/PAPER267/PAPER267.html

[2]

J. Carriere and R. Kazman,
WebQuery: searching and visualizing the Web through connectivity,
in: *Proc. of the 6th International World Wide Web Conference*, 1997, pp. 701–711,
http://www6.nttlabs.com/papers/PAPER96/PAPER96.html

[3]

J. Kleinberg,
Authoritative sources in a hyperlinked environment,
in: *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms,* 1998,
http://simon.cs.cornell.edu/home/kleinber/auth.ps;
also appeared as IBM Research Report RJ 10076, May 1997.

[4]

Y. Maarek, M. Jacovi, M. Shtalhaim, S. Ur, D. Zernik and I.Z. Ben Shaul,
WebCutter: a system for dynamic and tailorable site mapping,
in: *Proc. of the 6th International World Wide Web Conference*, 1997, pp. 713–722,
http://www6.nttlabs.com/papers/PAPER40/PAPER40.html

[5]

M. Marchiori,
The quest for correct information on the Web: hyper search engines,
in: *Proc. of the 6th International World Wide Web Conference*, 1997, pp. 265–274,
http://www6.nttlabs.com/papers/PAPER222/PAPER222.html

[6]

P. Pirolli, J. Pitkow, and R. Rao,
Silk from a sow's ear: extracting usable structures from the Web,
in: *CHI '96 Proceedings: Conference on Human Factors in Computing Systems,*, 1996, pp. 118-125,
http://www.acm.org/sigchi/chi96/proceedings/papers/Pirolli_2/pp2.html

URLs

- **BackRub**: http://backrub.stanford.edu/
- **CLEARweb**: http://www.clearweb.com
- **Dynamic Diagrams**: http://www.dynamicdiagrams.com/
- **InContext**: http://www.incontext.com/

- **MAPA**:http://www.dynamicdiagrams.com/products.htm#map
- **Mapuccino**: http://www.ibm.net.il/ibm_il/ibmhrl/webcutter/
- **Merzcom**: http://www.merzcom.com/
- **Merzscope**: http://www.merzcom.com/eng/merzscope/about.html
- **PageRank**: http://www-pcd.stanford.edu/~page/papers/pagerank/index.htm
- **Rankdex**: http://rankdex.gari.com/
- **SiteMapping**: http://backoffice.microsoft.com/products/features/SiteAnalyst/SiteServer_TechDetails.asp#sitemapping
- **WebAnalyzer**: http://www.incontext.com/products/analyze.html
- **WebSQL**: http://www.cs.toronto.edu/~websql/

Vitae

Krishna Bharat *is a member of the research staff at Digital Equipment Corporation's Systems Research Center. His research interests include Web content discovery and retrieval, user interface issues in automating tasks on the Web and speech interaction with hand-held devices. He received his Ph.D. in Computer Science from Georgia Institute of Technology in 1996, where he worked on tool and infrastructure support for building distributed user interface applications.*

Andrei Broder *has a B.Sc. from Technion, Israel Institute of Technology and a M.Sc. and Ph.D. in Computer Science from Stanford University. He is a member of the research staff at Digital Equipment Corporation's Systems Research Center in Palo Alto, California. His main interests are the design, analysis, and implementation of probabilistic algorithms and supporting data structures, in particular in the context of Web-scale applications.*

Monika R. Henzinger *received her Ph.D. from Princeton University in 1993 under the supervision of Robert E. Tarjan. Afterwards, she was an assistant professor in Computer Science at Cornell University. She joined the Digital Systems Research Center in 1996. Her current research interests are information retrieval on the Web, efficient algorithms and data structures, and program performance monitoring.*

Puneet Kumar *received his Ph.D. from Carnegie Mellon University in 1994. Afterwards, he joined the Digital Equipment Corporation's Systems Research Center as a member of the research staff. His current research interests are information retrieval on the Web, Internet appliances, compilers and file systems.*

Suresh Venkatasubramanian *is a Ph.D. Candidate in Computer Science at Stanford University working under the direction of Professors Rajeev Motwani and Jean-Claude Latombe. His research interests are in algorithm design/analysis, combinatorial geometry, and graph theory.*