WELCOME TO THE GATHERING PLACE

Computer Networks and ISDN Systems 30 (1998) 317–326

COMPUTER NETWORKS and ISDN SYSTEMS

The shark-search algorithm. An application: tailored Web site mapping

Michael Hersovici [a], Michal Jacovi [a,*], Yoelle S. Maarek [a], Dan Pelleg [b], Menachem Shtalhaim [a], Sigalit Ur [a]

[a] *IBM Haifa Research Laboratory, MATAM, Haifa 31905, Israel*
[b] *Department of Computer Science, Technion, Haifa, Israel*

Abstract

This paper introduces the "shark search" algorithm, a refined version of one of the first dynamic Web search algorithms, the "fish search". The shark-search has been embodied into a dynamic Web site mapping that enables users to tailor Web maps to their interests. Preliminary experiments show significant improvements over the original fish-search algorithm. © 1998 Published by Elsevier Science B.V. All rights reserved.

Keywords: Dynamic search; Site mapping; Resource discovery

1. Introduction and motivation

Web search services typically use a previously built index that is actually stored on the search service server(s). This approach is pretty efficient for searching large parts of the Web, but it is basically static. The actual search is performed on the server on all the data stored in the index. Results are not guaranteed to be valid at the time the query is issued (note that many search sites may take up to one month for refreshing their index on the full Web). In contrast, dynamic search actually fetches the data at the time the query is issued. While it does not scale up, dynamic search guarantees valid results, and is preferable to static search for discovering information in small and dynamic sub-Webs

One of the first dynamic search heuristics was the "fish search" [1], that capitalizes on the intuition that relevant documents often have relevant neighbours. Thus, it searches deeper under documents that have been found to be relevant to the search query, and stops searching in "dry" areas. However, the fish-search algorithm presents some limitations. Under time limit constraints, it does not discover "enough" relevant search directions. In this paper, we propose an improved and more "aggressive" version of that algorithm, that we call the "shark search" algorithm. The shark-search algorithm overcomes the limitations of its ancestor, by better estimating the relevance of neighbouring pages, even before they are accessed and analyzed.

We have embodied the shark-search algorithm into Mapuccino (previously known as WebCutter), a system for dynamically generating Web maps tailored to the user's interests, that was described at the previous WWW6 conference [3] and that is provided

* Corresponding author. E-mail: jacovi@haifa.vnet.ibm.com

as a free service to Web users at the **Mapuccino home page**[1].

Section 2 briefly reviews related work and describes in detail the fish-search algorithm and its limitations. Section 3 introduces the shark-search algorithm. Section 4 describes the Mapuccino system, a system for dynamic site mapping, in which the shark-search was embodied. Section 5 shows some experimental results obtained with Mapuccino when using these search heuristics and in particular shows the improvements achieved by the shark-search algorithm over the fish-search algorithm. Section 6 summarizes the contributions of this paper. Note that while the shark-search algorithm has been implemented in and for the Mapuccino system, i.e., for tailorable site mapping, it can in principle be used in other dynamic search systems.

2. 2. The fish-search algorithm

A typical example of dynamic search is **WebGlimpse**[2] [5] that allows users to dynamically search sub-areas of the Web predefined by a given "hop" parameter (i.e. depth in the Web viewed as a graph). The fish-search [1] proposes a more refined paradigm that can be explained by the following metaphor. It compares search agents exploring the Web to a school of fish in the sea. When food (relevant information) is found, fish (search agents) reproduce and continue looking for food, in the absence of food (no relevant information) or when the water is polluted (poor bandwidth), they die. The key principle of the algorithm is the following: it takes as input a seed URL and a search query, and dynamically builds a priority list (initialized to the seed URL) of the next URLs (hereafter called nodes) to be explored. At each step the first node is popped from the list and processed. As each document's text becomes available, it is analyzed by a scoring component evaluating whether it is relevant or irrelevant to the search query (*1–0* value) and, based on that score, a heuristic decides whether to pursue the exploration in that direction or not: Whenever a document source is fetched, it is scanned for links. The nodes pointed to by these links (denoted "children") are each assigned a depth value. If the parent is relevant, the depth of the children is set to some predefined value. Otherwise, the depth of the children is set to be one less than the depth of the parent. When the depth reaches zero, the direction is dropped and none of its children is inserted into the list.

Children whose depth is greater than 0 are inserted in the priority list according to the following heuristics:

(1) the first $\alpha * width$ children (with α a predefined constant greater than *1*) of a relevant node are added to the head of the list;
(2) the first *width* children of an irrelevant node are added to the list right after the last child of a relevant node;
(3) the rest of the children are added to the tail (to be handled only if time permits).

Note that this is equivalent to affecting to each child what we term a "potential score" set to *1* in case 1 above, to *0.5* in case 2, and to *0* in case 3, and inserting the child at the right location in the priority list sorted by that potential score. To facilitate the comparison with the shark-search algorithm described later, we use this equivalent representation in the following. Figure 1 is a description of the fish-search algorithm (in a simplified form, omitting some optimization details for the sake of the generality).

The fish-search algorithm, while being attractive because of the simplicity of its paradigm, and its dynamic nature, presents some limitations.

First, it assigns a relevance score in a discrete manner (*1* for relevant, *0* or *0.5* for irrelevant) using primitive string- or regular-expression match. More generally, the key problem of the fish-search algorithm is the very low differentiation of the priority of pages in the list. When many documents have the same priority, and the crawler is restricted to a fairly short time, arbitrary pruning occurs — the crawler devotes its time to the documents at the head of the list. Documents which are further along the list whose scores may be identical to some further along may be more relevant to the query. In addition, cutting down the number of addressed children by using the *width* parameter is arbitrary, and may result in loosing valuable information. Clearly, the main issue that needs to be addressed is a finer grained

[1] http://www.ibm.com/java/mapuccino
[2] http://glimpse.cs.arizona.edu/webglimpse/index.html

```
• Get as Input parameters, the initial node, the width (width), depth (D) and size (S) of the desired graph to be explored, the
  time limit, and a search query
• Set the depth of the initial node as depth = D, and Insert it into an empty list
• While the list is not empty, and the number of processed nodes is less than S, and the time limit is not reached
  (1) Pop the first node from the list and make it the current node
  (2) Compute the relevance of the current node
  (3) If depth > 0:
      (1) If current_node is irrelevant
          Then
            ■ For each child, child_node, of the first width children of current_node
                ■ Set potential_score(child_node) = 0.5
            ■ For each child, child_node, of the rest of the children of current_node
                ■ Set potential_score(child_node) = 0
          Else
            ■ For each child, child_node, of the first (a * width) children of current_node
              (where a is a pre-defined constant typically set to 1.5)
                ■ Set potential_score(child_node) = 1
            ■ For each child, child_node, of the rest of the children of current_node
                ■ Set potential_score(child_node) = 0
      (2) For each child, child_node, of current_node,
            ■ If child_node already exists in the priority list,
              Then
                (1) Compute the maximum between the existing score in the list to the newly computed potential score
                (2) Replace the existing score in the list by that maximum
                (3) Move child_node to its correct location in the sorted list if necessary
              Else Insert child_node at its right location in the sorted list according to its potential_score value
      (3) For each child, child_node, of current_node,
            ■ Compute its depth, depth(child_node), as follows:
              (1) If current_node is relevant,
                  Then Set depth(child_node) = D
                  Else depth(child_node) = depth(current_node) − 1
              (2) If child_node already exists in the priority list
                  Then
                    (1) Compute the maximum between the existing depth in the list to the newly computed depth
                    (2) Replace the existing depth in the list by that maximum.
• EndWhile
```

Fig. 1. The fish-search algorithm.

scoring capability. This is problematic because it is difficult to assign a more precise potential score to documents which have not yet been fetched.

3. The shark-search algorithm

We propose several improvements to the original fish-search algorithm in order to overcome these limitations. They result in a new algorithm, called the "shark search" algorithm, that while using the same simple metaphor, leads to the discovery of more relevant information in the same exploration time.

One immediate improvement is to use, instead of the binary (relevant/irrelevant) evaluation of document relevance, what we will call hereafter a *similarity engine* in order to evaluate the relevance of documents to a given query. Such an engine analyzes two documents dynamically and returns a "fuzzy" score, i.e., a score *between 0* and *1* (*0* for no similarity whatsoever, *1* for perfect "conceptual" match) rather than a binary value. A straightforward method for building such an engine is to apply the usual vector space model [6]. It has been shown that these kinds of techniques give better results than simple string- or regular-expression matching [7, p. 306].

The similarity algorithm can be seen as orthogonal to the fish-search algorithm. We will assume in the rest of the paper that such an engine is available and that for any pair query, document, ***(q,d)***, it returns a similarity score ***sim(q,d)*** between *0* and *1*.

This first improvement has a direct impact on the priority list. We use a "fuzzy" relevance score, giving the child an *inherited score*, thus preferring the children of a node that has a better score. This information can be propagated down the descendants chain, thus boosting the importance of the grandchildren of a relevant node over the grandchildren of an irrelevant node. The children of an irrelevant node use their parent's inherited score for calculating their own inherited score — by multiplying it by some decay factor δ, whose role is comparable to Marchiori's "fading" factor [4]. Thus, suppose documents X and Y were analyzed and that X has higher score. Suppose further that the children of both X and Y have a null score, and the algorithm now has to select the most promising of their grandchildren. Since both their scores are multiplied by δ^2, the shark search chooses X's grandchildren first (which seems intuitively correct). Stepping further away from X, into the great-grandchildren zone, would bring Y's descendants back into consideration, given an appropriate selection of δ. This continuous valued behavior of score inheritance is much more natural than the Boolean approach of the fish-search.

A more significant improvement consists of refining the calculation of the potential score of the children not only by propagating ancestral relevance scores deeper down the hierarchy, but also by making use of the meta-information contained in the links to documents. We propose to use the hints that appear in the parent document, regarding a child. The anchor text of the link is the author's way to hint as to the subject of the linked document. A surfer on the Web, encountering a page with a large amount of links, will use the anchor text of the links in order to decide how to proceed. Some automated tools (see [2]) also take advantage of that information. This approach however can fail in poorly styled HTML documents, in which anchor texts consist only of "click here" or "jump there", or when the anchor information is a picture without ALT information. To remedy this problem we suggest using the close textual context of the link, and combining the information extracted from it with the information extracted from the anchor text. To reduce the risk of mistakenly giving a high potential score to a node due to a textual context that actually belongs to another node linked in the same paragraph (context boundaries are difficult to identify), we suggest a small fix. If a node has a relevant anchor text, the score of the textual context is set to the maximum (value of 1), thus giving it an edge over neighbouring links. We claim that a policy that selects the children with the most promising anchor and anchor context information is preferable to arbitrarily selecting the *first* set of children.

These heuristics are so effective in increasing the differentiation of the scores of the documents on the priority list, that they make the use of the width parameter redundant — there is no need to arbitrarily prune the tree. Therefore, no mention is made of the the *width* parameter in the shark-search algorithm that is more formally described in Fig. 2 (note that only the delta from the fish-search algorithm is given).

4. One embodiment of the shark-search algorithm: Mapuccino

Mapuccino (previously known as WebCutter [3]) is a tool for dynamically generating site maps tailored to the user's interests (expressed in free text). The basic principle behind Mapuccino's approach is the following: Typical site mapping tools start from a seed URL, and crawl the neighbouring Web space breadth first until either a certain time has elapsed, a given number of nodes or given depth and width upper bounds are reached. In contrast, by integrating search heuristics such as the fish- or shark-search algorithms in the crawling process so as to go more deeply in directions where relevant information was found, we allow users to generate maps tailored to their interests.

Tailored maps have "longer arms" in relevant directions. They display not only pages actually relevant to the topic of interest (highlighted in blue, shaded lighter or darker according to the relevance in Fig. 3) but also their immediate neighbourhood (not highlighted, i.e., coloured in white) so as to show relevant information in context.

```
In the fish-search algorithm, replace step 3.1, for computing the child's potential score, with the following:
(1) Compute the inherited score of child_node, inherited_score(child_node), as follows:
    • If relevance(current_node) $> 0$ (the current node is relevant)
      Then inherited_score(child_node) $= \delta *$ sim(q,current_node)
      where $\delta$ is a predefined decay factor.
      Else inherited_score(child_node) $= \delta *$ inherited_score(current_node)
(2) Let anchor_text be the textual contents of the anchor pointing to child_node, and anchor_text_context, the textual context of
    the anchor (up to given predefined boundaries)
(3) Compute the relevance score of the anchor text as anchor_score = sim(q,anchor_text)
(4) Compute the relevance score of the anchor textual context as follows:
    • If anchor_score $> 0$,
      Then anchor_context_score = 1
      Else anchor_context_score = sim(q,anchor_text_context)
(5) Compute the score of the anchor, that we denote neighbourhood_score as follows:
    neighbourhood_score $= \beta *$ anchor_score $+ (1 - \beta) *$ anchor_context_score
    where $\beta$ is a predefined constant
(6) Compute the potential score of the child as
    potential_score(child_node) $= \gamma *$ inherited_score(child_node) $+ (1 - \gamma) *$ neighbourhood_score(child_node)
    where $\gamma$ is a predefined constant.
```

Fig. 2. The shark-search algorithm.

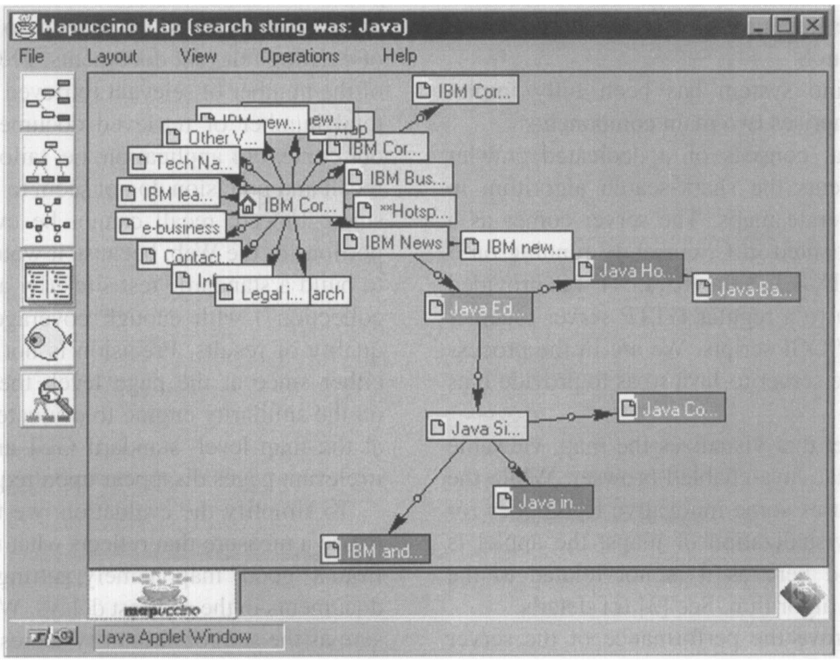


Fig. 3. Tailored map in "circle" view.

In a typical scenario, the user accesses a Mapuccino server via his/her Web browser, specifies what site (or seed URL) s/he wishes to map, how long s/he is ready to wait, and more importantly what topics s/he is more interested in. The server then works in three steps:

(1) It crawls the site going more deeply in directions where relevant information is found.
(2) It identifies a graph of Web pages.
(3) It returns the graph back to the user when the allocated time has expired, together with a Java ap-

plet that visualizes the map on the user's browser (other alternate views, besides the circle-view shown here, are provided via button click on the toolbar).

The pages relevant to the user's interests are shown in context together with pages that might not be relevant but are hyperlinked to them. Thus, if a user generated a map tailored to the topic "Java", s/he may see a highlighted page about "Java education", which itself is related to some pages dealing with education on the object-oriented paradigm. While the latter pages are not directly related to the original user's query, and thus are not highlighted in the map, they can still present some interest. The context as defined by the Web authors thus brings additional information.

Note that the purpose of this tool is not to simply search for information in a site, but rather to generate a global view of a Web area where the user knows that relevant information can be found. In other terms, the prime intention of the user is to browse rather than to search.

The Mapuccino system has been fully implemented, and comprises two main components:

(1) A server that consists of a dedicated crawler that implements the shark-search algorithm in order to generate maps. The server comes as a DLL implemented in C, currently running on 3 platforms: AIX 4.1, Solaris 2.4, NT 4.0, provided as an add-on to a regular HTTP server together with a set of CGI scripts. We are in the process of porting the server to Java so as to provide it as a servlet.

(2) A Java applet that visualizes the map, viewable by any regular Java-enabled browser. While the Java applet uses some innovative techniques for layout and visualization of maps, the applet is not described here as it is not related to the shark-search algorithm. See [3] for details.

In order to improve the performance of the server in terms of communication, in our implementation of the algorithm we used multiple communication channels (sockets) as opposed to the original fish-search algorithm implementation that only used one communication channel. Using multiple sockets significantly improved the performance of the server. From empirical tests made against a single HTTP server, we found that using 5 sockets resulted in the best overall transfer rate. The server is highly configurable, and works with most classical HTTP servers. It respects the Robots Exclusion Protocol, the de facto standard for limiting Web spiders access within sites. Maps up to a size of 30,000 nodes were successfully generated by the server (required time depending on the bandwidth at the time of execution).

5. Experimental results

5.1. Evaluation measure

In order to evaluate the effectiveness of the shark-search algorithm as compared to the original fish-search, we first thought of using the traditional information retrieval criteria for evaluating retrieval effectiveness of search engines, namely *recall* and *precision*. Recall is defined as the ratio of the number of relevant retrieved documents to the total number of existing relevant documents. Precision is the ratio of the number of relevant retrieved documents to the total number of retrieved documents. However, in our context of application, i.e. tailored site mapping, recall and precision do not seem to be adequate measures. Indeed, recall cannot be evaluated on large portions of the Web, because it would be too difficult to build a standard "test site" (by analogy with "test collection") with enough coverage to evaluate the quality of results. Precision is not a good indicator either since at the page level, the algorithm relies on the similarity engine to evaluate relevance, while at the map level, standard GUI artifacts can make irrelevant pages disappear upon request.

To simplify the evaluation, we therefore propose to use a measure that reflects what most users expect from a "good" map: namely, getting as many relevant documents in the shortest delays. We define this measure as the sum of similarity scores of all documents forming the maps, which we will refer to as the *"sum of information"* measure in the rest of this paper. Note that we do not use an average measure (for instance dividing the sum by the total number of nodes in the map) since irrelevant documents that were identified in the same process do not hurt the quality of the map. On the contrary, they give context information and can be pruned from the map if so desired.

So in summary, if we define as ***sim(d,q)***, the similarity score (between *0* and *1*) between any document ***d*** in the map and query ***q*** directing the mapping, and ***M*** the set of documents in a given map, we have

$$\textit{sum\_of\_information}(M,q) = S_{(d \text{ in } M)}\, \textit{sim}(d,q)$$

Note that ***sim(d,q)*** is the actual similarity score as computed by the similarity engine for documents whose contents was fetched by the crawler. Potential scores, which are speculative, do not affect this value.

5.2. Selection of parameters

The shark-search algorithm makes use of 3 predefined parameters, β, γ and δ (see Fig. 2). We tried to find a combination of the parameters which maximizes the expected score of a search set. Several experiments were made, using different search queries and starting from different URLs. The "depth" value was always set to *3*. The time-out period ranged from one to five minutes, but was fixed for the same test case. The number of documents to be scanned was chosen such that the time-out constraint will be the one stopping the algorithm. Several combinations of values for β, γ and δ were tried, with 10 to 18 combinations for each experiment.

Coming to include fish-search as a control group in the comparison tests with the shark-search algorithm, we decided to slightly refine the tested fish-search component by using the same retrieval engine as for the shark-search component so as not to bias the experiments and unfairly penalize the fish-search algorithm.

Throughout the experiments, the shark-search runs obtained significantly higher *sum-of-information* values than the fish-search algorithm. In average, the best results were obtained at $\delta = 0.5$, $\beta = 0.8$ and $\gamma = 0$, as the results were systematically in the top three *sum-of-information* values (note that it happened that the maximum was reached a few times at $\gamma = 1$, and once at $\gamma = 0.5$).

5.3. Some examples

Table 1 shows the performance of the shark-search against that of the fish-search, for four test cases. We see that the shark-search performed between 1.5 and 3 times better than its ancestor.

Note that these test cases having been designed experimentally (rather than formally over a statistically significant sample, and using testing guidelines procedure), they should be considered as a positive indicator of the effectiveness of the algorithm rather than as formal results.

We used three variations of the Mapuccino engine, (all using the same similarity engine) to generate the maps below. The seed in all examples was the CNN home page (depicted by a small house icon), using "San Francisco 49ers" as search query for directing the crawling process. Relevant pages are highlighted in blue (irrelevant ones in white). The darker the shade of blue, the more relevant the document is. The same amount of time was affected in all three cases.

Case 1: fish-search algorithm (with similarity engine). The map in Fig. 4 shows that 3 directions were identified by the fish-search algorithm from the CNN home page, and one got explored somewhat

Table 1
Performance results in the sum-of-information metric

Test case	Fish-search sum of info	Shark-search sum of info	Improvement ratio
query: *"computational biology course"* **seed URL**: *http://www.cs.technion.ac.il*	281	324	1.15
query: *" Global survey Mars exploration sojourner"* **seed URL**: *http://www.cnn.com*	71	189	2.66
query: *"breast feeding nutrition"* **seed URL**: *http://www.parentsplace.com*	316	645	2.04
query: *"semi-definite programming and Cholesky decomposition"* **seed URL**: *http://www.mcs.anl.gov/otc/Guide/*	34	126	3.71

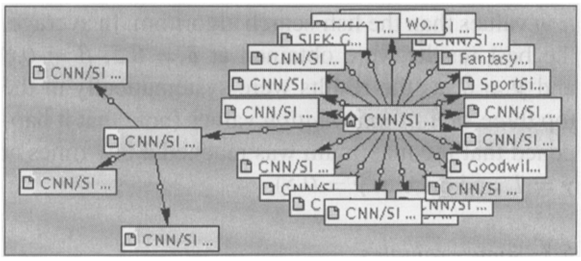


Fig. 4. Tailored map (fish-search algorithm).

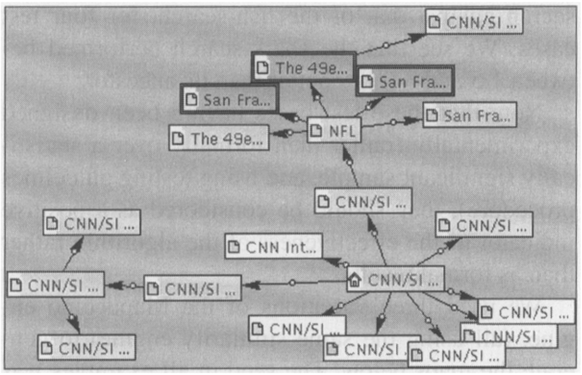


Fig. 5. Tailored map (partial shark-search algorithm).

deeper in the given time span. The total of relevant pages thus identified amounts to 7.

Case 2: shark-search algorithm without context analysis. As seen in Fig. 5, the same directions were explored at the first level, but thanks to the queue prioritization, the server did not waste time exploring irrelevant pages identified at the first level in the previous case, and could go deeper in two relevant directions, identifying more relevant pages as indicated by the darker shade of blue. A total of 14 relevant documents was found, and 3 of them were particularly relevant (no document with this degree of relevance was found in the previous case).

Case 3: complete shark-search algorithm. The tendency shown in the previous example gets more pronounced. The first-level descendant that seemed to induce the best direction (after the fact) is immediately identified as a highly relevant page (darker blue), see Fig. 6, and more sub-directions are identified in the same time span. The number of relevant pages found increased to 17, 4 of them being highly relevant.

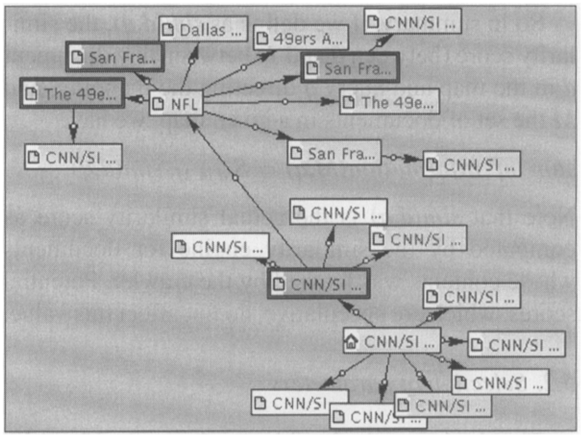


Fig. 6. Tailored map (complete shark-search algorithm).

6. Conclusion

In this paper, we have proposed an improved version of one of the first dynamic search algorithms for the Web, the "fish search" algorithm. Our more aggressive shark-search algorithm overcomes some limitations of the fish search by analyzing the relevance of documents more precisely and, more importantly making a finer estimate of the relevance of neighbouring pages before they are actually fetched and analyzed. These improvements enable to save precious communication time, by fetching first documents that are most probably relevant or leading to relevant documents, and not wasting time on garden paths. We implemented both the fish- and the shark-search algorithms in a dynamic tailorable site mapping tool, **Mapuccino**[3], and experimentally verified that significant improvements were achieved in terms of the total of relevant information discovered in the same small amount of time. We believe that the shark-search algorithm can thus advantageously replace its ancestor in applications that require dynamic and precise searches in limited time spans.

Acknowledgments

We are grateful to David Notkin for his valuable comments and suggestions in the writing of this pa-

[3] http://www.ibm.com/java/mapuccino

per. This research was done while Dan Pelleg was an extern student at the IBM Haifa Research Laboratory.

References

[1] P. De Bra, G.-J. Houben, Y. Kornatzky, and R. Post, Information retrieval in distributed hypertexts, in: *Proceedings of RIAO'94, Intelligent Multimedia, Information Retrieval Systems and Management*, New York, NY, 1994.

[2] M. Iwazume, K Shirakami, K. Hatadani, H. Takeda and T. Nishida, IICA: An ontology-based Internet navigation system, in: *AAAI-96 Workshop on Internet Based Information Systems*, 1996.

[3] Y.S. 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*, Santa-Clara, April 1997, http://proceedings.www6conf.org .

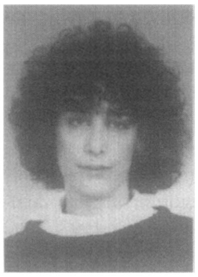
[4] M. Marchiori, The quest for correct information on the Web: Hyper search engines, in: *Proc. of the 6th International World Wide Web Conference*, Santa-Clara, April 1997, http://proceedings.www6conf.org

[5] U. Manber, M. Smith and B. Gopal, WebGlimpse: combining browsing and searching, in: *Proceedings of the Usenix Technical Conference*, Jan. 6–10, 1997, 997; WebGlimpse paper, ftp://ftp.cs.arizona.edu/people/udi/webglimpse.ps.Z and WebGlimpse home page, http://glimpse.cs.arizona.edu/webglimpse/index.html

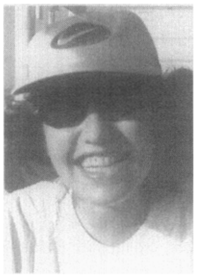
[6] G. Salton and M.J. McGill, *Introduction to Modern Information Retrieval*, Computer Series, McGraw-Hill, New York, NY, 1983.

[7] G. Salton, *Automatic Text Processing, the Transformation, Analysis and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.

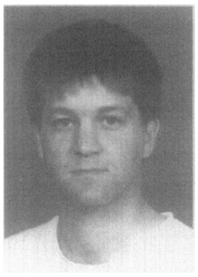
Michael Herscovici is a Research Staff Member at the IBM Haifa Research Lab in Haifa, Israel and belongs to the "Information Retrieval and Organization" Group. His research interests include Internet applications and parsing techniques. Mr. Herscovici will receive his B.Sc. in Computer Science from the Technion, Israel institute of technology in Haifa, in 1998. He joined IBM in 1997 and has since worked on the dedicated robot component of Mapuccino, a Web site mapping tool.



Michal Jacovi is a Research Staff Member at the IBM Haifa Research Lab in Haifa, Israel, and belongs to the "Information Retrieval and Organization" Group. Her research interests include Internet applications, user interfaces, and visualization. She received her M.Sc. in Computer Science from the Technion, Haifa, Israel, in 1993. Ms. Jacovi has joined IBM in 1993, and worked on several projects involving user interfaces and Object Oriented, some of which have been published in journals and conferences. Since the emergence of Java, she has been involved in the conception and implementation of Mapuccino, a Web site mapping tool, written in Java, that is being integrated into several IBM products.



Yoelle S. Maarek is a Research Staff Member at the IBM Haifa Research Lab in Haifa, Israel and manages the "Information Retrieval and Organization" Group that counts about 10 staff members. Her research interests include information retrieval, Internet applications, and software reuse. She graduated from the "Ecole Nationale des Ponts et Chaussees", Paris, France, as well as received her D.E.A (graduate degree) in Computer Science from Paris VI University in 1985. She received a Doctor of Science degree from the Technion, Haifa, Israel, in January 1989. Before joining IBM Israel, Dr. Maarek was a research staff member at the IBM T.J. Watson Research Center for about 5 years. She is the author of Guru, an advanced information retrieval system, widely used within IBM, and lead the team that conceived and implemented Mapuccino, a Web site mapping tool, written in Java, that is being integrated into several IBM products. She has published over 20 papers in referred journals and conferences.



Dan Pelleg received his B.A. from the Department of Computer Science, Technion, Haifa, Israel, in 1995. He is currently an M.Sc. student in the same department, finishing his Master's thesis, "Phylogeny Approximation via Semidefinite Programming". His research interests include computational biology, combinatorial optimization and Web-based software agents. During the summer of 1997, Dan worked as an extern student in IBM Haifa Research Laboratory, where he focused on the Shark-search algorithm.

Menachem Shtalhaim is a Research Staff Member at the IBM Haifa Research Lab in Haifa, Israel and belongs to the "Information Retrieval and Organization" Group. His research interests include Internet applications, communication protocols and heuristic algorithms. Mr. Shtalhaim joined IBM in 1993, and worked on several projects involving morphological analysis tools, Network Design and analysis tool (IBM product NetDA/2) and the AS400 logical file system layer. In the past, Mr. Shtalhaim have worked on medical diagnostic systems. He is the author of the dedicated robot component of Mapuccino, a Web site mapping tool.

Sigalit Ur is a Research Staff Member at the IBM Haifa Research Lab in Haifa, Israel, working on Mapuccino, a Web site mapping tool, written in Java, that is being integrated into several IBM products. She received a Master of Science degree in Intelligent Systems from the University of Pittsburgh in 1993. Before joining IBM, Ms. Ur was involved in projects in a wide variety of fields, including data processing, databases, cognitive science, multi-agent planning and image processing, some of which have been published in journals and conferences.