Reducing Human Interactions in Web Directory Searches

ORI GERSTEL
Cisco
SHAY KUTTEN
Technion
EDUARDO SANY LABER
PUC-Rio
RACHEL MATICHIN and DAVID PELEG
The Weizmann Institute of Science
ARTUR ALVES PESSOA
UFF
and
CRISTON SOUZA
PUC-Rio

Consider a website containing a collection of webpages with data such as in Yahoo or the Open Directory project. Each page is associated with a weight representing the frequency with which that page is accessed by users. In the tree hierarchy representation, accessing each page requires the user to travel along the path leading to it from the root. By enhancing the index tree with additional edges (hotlinks) one may reduce the access cost of the system. In other words, the hotlinks reduce the expected number of steps needed to reach a leaf page from the tree root, assuming that the user knows which hotlinks to take. The *hotlink enhancement* problem involves finding a set of hotlinks minimizing this cost.

This article proposes the first exact algorithm for the hotlink enhancement problem. This algorithm runs in polynomial time for trees with logarithmic depth. Experiments conducted with

S. Kutten and D. Peleg were supported in part by a grant from the Israel Science Foundation.
Authors' addresses: O. Gerstel, Cisco, 170 West Tasman Drive, San Jose, CA 95134; S. Kutten (corresponding author), Information Systems Group, Faculty of Industrial Engineering and Management, Technion, Haifa 32000, Israel; email: kutten@ie.technion.ac.il; E. S. Laber, C. Souza, Departamento de Informatica, PUC-Rio, Rio de Janeiro, Brazil; email: laber@inf.puc-rio.br; cristonsouza@yahoo.com.br; R. Matichin D. Peleg, Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, 76100 Israel; email: {rachelm,peleg}@wisdom.weizmann.ac.il; A. A. Pessoa, Departamento de Engenharia de Produo, UFF, Niteri—RJ, Brazil; email: artur@producao.uff.br.

real data show that significant improvement in the expected number of accesses per search can be achieved in websites using this algorithm. These experiments also suggest that the simple and much faster heuristic proposed previously by Czyzowicz et al. [2003] creates hotlinks that are nearly optimal in the time savings they provide to the user.

The version of the hotlink enhancement problem in which the weight distribution on the leaves is unknown is discussed as well. We present a polynomial-time algorithm that is optimal for any tree for any depth.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Distributed data structures, trees*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Theory and methods*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Hotlink, hotlist, hyperlink, directory tree, algorithms

ACM Reference Format:

Gerstel, O., Kutten, S., Laber, E. S., Matichin, R., Peleg, D., Pessoa, A. A., and Souza, C. 2007. Reducing human interactions in web directory searches. ACM Trans. Inform. Syst. 25, 4, Article 20 (October 2007), 28 pages. DOI = 10.1145/1281485.1281491 http://doi.acm.org/ 10.1145/1281485.1281491

1. INTRODUCTION

1.1 Motivation

Finding desired information in a large and diverse database is a complex task. When such a function is needed in a chaotic and large data collection such as the World Wide Web, such a function becomes even harder, yet crucial. There are two basic ways to handle information-finding in such a collection. One is a "flat" approach which views the information as a nonhierarchical structure and provides a query language to extract the relevant data from the database. An example of this approach on the web is the Google search engine [Google 2007]. The other method is based on a hierarchical index to the database according to a taxonomy of categories. Examples of such indices on the web are Yahoo [2007] and the Open Directory Service [DMOZ 2007].

An advantage of the flat approach over the hierarchical is that the number of human operations required to find a desired piece of information is much lower (if the right query is used). By contrast, in the hierarchical approach it is necessary to traverse a path in the taxonomy tree from the root to the desired node in the tree. Human engineering considerations further aggravate this problem, since it is very hard to choose an item from a long list (typical convenient numbers are 7–10). Thus, the degree of the taxonomy tree should be rather low and its average depth, therefore, high.

Another problem in the hierarchical approach is that the depth of an item in the taxonomy tree is not based on the access pattern (see, e.g., Attardi et al. [1998]). As a result, items which have very high access frequency may require long access paths each time they are needed, while those which are "unpopular" may still be very accessible in the taxonomy tree. It is desirable to find a solution that does not change the taxonomy tree itself, since this taxonomy is likely to be meaningful and useful for the user.

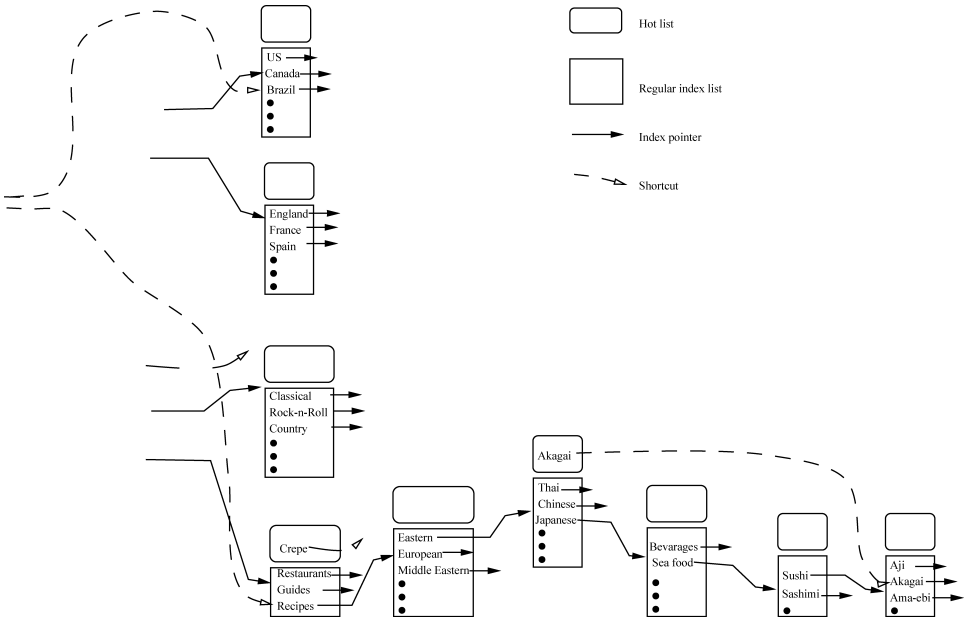

Fig. 1. An example for an index tree enhanced with hotlinks.

A partial solution to this problem is currently used in the web, and consists of a list of "hot" pointers which appears in the top level of the index tree and leads directly to the most popular items. We refer to a link from a hotlist to its destination as a *hotlink*. This approach is not scalable in the sense that only a small number of items can appear in such a list.

In the current article we study the generalization of this "hotlist" proposed by Bose et al. [2000]. This approach allows us to have such lists in multiple levels in the index tree, not just the top level. The resulting structure is termed a *hotlink-enhanced index structure* (or *enhanced structure* for short).

An example of such an index tree and the corresponding enhanced structure, augmented with multiple hotlists, appears in Figure 1. In this example, suppose that some browsing user is interested in a recipe for a specific type of Japanese sushi called *akagai*. In the original tree index (without hotlinks) the user would have to start at the main page on the top level, go to the "Entertainment" page on the second level, from there to "Food", "Recipes", "Eastern food", "Japanese food", "Sea food", "Sushi", and finally find "Akagai"—a total of seven accesses and list browsing operations. With the hotlinks depicted in the figure, however, the user sees "Food" already at the top level. Since the user is looking for a recipe, that user would use the hotlink. Since the hotlink at the "Food" node is not currently relevant to that user, the user would use a regular link to "Recipes" and then to "Eastern". At this point the user would encounter a hotlink to "Akagai" and find the requested information using only four hops.

The decision concerning the hotlinks to be added is based on the statistics of visited items in the index. The goal is to minimize the expected number of links one has to follow from the root to an item. It is, therefore, possible to

consider static systems where the hotlinks do not get updated often, as well as dynamic systems which follow the access pattern and dynamically change hotlinks on-the-fly. In this article, we consider the former, simpler system.

There are many applications for such hotlink-enhanced index structures. A partial list includes:

—a web index (such as Yahoo), with multilevel hotlists in which the access statistics is influenced by accesses of all users to various sites;

—a personalized web index in which the browser records personalized statistics;

—large library index systems.

—Application menu trees are currently designed by the application developer, or statically customized to the needs of a user. By adding hotlists, the application can learn the usage pattern of the user and adjust to changes in this pattern.

—It is even possible to use the preceding idea in file systems, in which files in the static tree structure of the file system can be augmented by links to frequently accessed subdirectories or files.

1.2 The Problem

An index system is based on a fixed tree which classifies the data items in a hierarchical manner. For the sake of simplicity, we assume that data resides only in leaves of the tree (in Section 2.4 we discuss how this assumption can be removed). To this tree, hotlinks are added based on access statistics to the data items, yielding the hotlink-enhanced index structure.

When searching for an item, the user starts from the root of the enhanced structure and advances along tree edges and hotlinks towards the required destination. The original index tree contains a unique path leading from the root of the tree to the desired leaf. An implicit assumption underlying the common hierarchical approach is that at any node along the search in the tree, the user is able to select the correct link leading towards the desired leaf. This does not necessarily mean that the user knows the tree topology, but rather that the user has some general knowledge about the domain, and the links the user finds at any node reflect some natural partitioning of this domain. Thus, when seeing several tree edges and hotlinks at a node, the user is capable of selecting the right link downwards in the tree.

Once hotlinks are added the situation becomes more complex, as the resulting hotlink-enhanced index structure is no longer a tree, but a directed acyclic graph (DAG) with multiple alternative paths for certain destinations. Again, an underlying assumption as the basis of the hotlink idea is that when faced with a hotlink in the current page, the user will be able to tell whether this hotlink may lead to a closer point on the path to the desired destination. However, the considerations that led to adding the various hotlinks to the tree are not known to the user. Thus, when at some node, the user can know only the hotlinks emanating from the current node (or, in the best case, also those hotlinks emanating from previous nodes the user visited on the way from the

root to the current node). In particular, the user cannot know whether any hotlinks emanate from descendants of the current node, nor where they lead.

The preceding discussion implies that any approach taken for designing a hotlink-enhanced index structure must take into account certain assumptions regarding the user's search policy. There could be a number of different models concerning the particular choices taken by the user. At the extreme lies a natural model that is probably too strong to be used in reality. This model captures situations where the user somehow knows the topology of the enhanced structure. Henceforth, we refer to this model as the clairvoyant user model, which is based on the following assumption.

In the clairvoyant user model, the user starts at the root of the tree and at each node in the enhanced structure can infer from the link labels which tree edges or hotlinks available at the current page are on a shortest path (in the enhanced structure) to the desired destination. The user always chooses that link.

In contrast, the model considered here is based on the assumption that the user does not have this knowledge. This forces the user to deploy a *greedy* strategy. In this user model, the user starts at the root of the tree and, at each node in the enhanced structure, can infer from the link labels which tree edges or hotlinks available at the current page lead to a page that is closest *in the original tree structure* to the desired destination. The user always chooses that link.

In this model, whenever looking at the page at node v in the hotlink-enhanced structure, the user is aware of all the tree edges and hotlinks in that page, but the user's knowledge about the rest of the tree corresponds only to the logical partitioning of the domain, namely, the original tree structure. In other words, the user is not aware of other hotlinks that may exist from other pages in the tree. This means that the user's estimate for the quality of a tree edge or hotlink leading from v to some node u is based solely on the height of u in the original tree (or equivalently, the distance from u to the desired destination in the tree).

An important implication of this discussion is that following the greedy strategy does not necessarily lead to an optimal path in the hotlink-enhanced index structure. For example, the enhanced structure presented in Figure 1 contains a hotlink from the main page (the root) to "Food". This hotlink will be taken by any user searching for "Akagai". If we now add a new hotlink from the "Entertainment" page directly to "Akagai", we create a better path of merely two hops from the main page to "Akagai", but this path will effectively never be used due to the greedy strategy. Recall that the logic behind the greedy assumption is that the user has no way of anticipating in advance the existence of improving hotlinks farther down the road after following a seemingly suboptimal link. Hence, the most reasonable policy for the user is to take, at any stage, the tree edge or hotlink that leads as much as possible to the desired destination.

This article addresses the optimization problem faced by the index designer, namely, to find a set of hotlinks that minimizes the expected number of links (either tree edges or hotlinks) traversed by a greedy user from the root to a leaf.

More formally, given a tree T with n nodes representing an index, a hotlink is an edge that does not belong to the tree. The hotlink starts at some node v and ends at (or *leads* to) some node u that is a descendant of v. (One may possibly consider a different model which allows to have the hotlinks from a node v to a nondescendant node u residing in another subtree; in our model, however, such hotlinks will never be used, due to the greedy assumption.) We assume without loss of generality that u is not a child of v. Each leaf x of T has a weight $p(x)$, representing the proportion of the user's visits to that leaf compared with the total set of user's visits. Hence, if normalized, $p(x)$ can be interpreted as the probability that a user wants to access the leaf x. Another parameter of the problem is an integer K, specifying an upper bound on the number of hotlinks that may start at any given node (there is no a priori limit on the number of hotlinks that lead to a given node).

Let S be a set of hotlinks constructed on the tree (obeying the bound of K outgoing hotlinks per node) and let $D_S(v)$ denote the greedy path (including hotlinks) from the root to node v. The expected number of operations needed to get to an item is

$$f(T, p, S) = \sum_{v \in Leaves(T)} |D_S(v)| \cdot p(v).$$

The problem of optimizing this function is referred to as the *hotlink enhancement* problem. Two different *static* problems arise, according to whether the probability distribution p is known to us in advance. Assuming a *known distribution*, our goal is to find a set of hotlinks S which minimizes $f(T, p, S)$ and achieves the optimal cost

$$\hat{f}(T, p) = \min_S \{ f(T, p, S) \} .$$

Such a set is termed an *optimal* set of hotlinks. On the other hand, under the *unknown distribution* assumption, the worst-case expected access cost for a tree T with a set of hotlinks S is

$$\tilde{f}(T, S) = \max_p \{ f(T, p, S) \} ,$$

and our goal is to find a set of hotlinks S minimizing $\tilde{f}(T, S)$ and achieving the optimal cost

$$\tilde{f}(T) = \min_S \{ \tilde{f}(T, S) \} .$$

For the latter problem, there exists an equivalent formulation, independent of the probability distributions, based on the following observation.

LEMMA 1.1. *For every tree T and hotlink function S,*

$$\tilde{f}(T, S) = \max_{v \in Leaves(T)} \{ |D_S(v)| \}.$$

COROLLARY 1.2. *For every tree T,*

$$\tilde{f}(T) = \min_S \left\{ \max_{v \in Leaves(T)} \{ |D_S(v)| \} \right\}.$$

1.3 Related Work

A considerable amount of research was invested into attempting to predict the next hyperlink clicked, or the eventual target of the user, and trying to aid the user in reaching it. For example, in many new Microsoft programs such as MS Word, a menu (e.g., file menu) will show near the top of the links that the user has used recently, while many other links may not be shown at all unless the user clicks some special symbol in the menu. Web Watcher [Armstrong et al. 1995] attempts to predict the link that users will follow on a particular page as a function of their specified interests. The link that WebWatcher predicts the user is likely to follow will be highlighted graphically and duplicated at the top of the page. The AVANTI project [Fink et al. 1996] attempts to guess the final destination of the user, and present a link to that guessed final destination.

For the clairvoyant user model, not used in the current work, a proof of NP-hardness for adding hotlinks on DAGs was presented in Bose et al. [2000] by a reduction from the problem of exact cover by 3-sets, which is known to be NP-complete. An interesting analogy was presented for the clairvoyant user model between the problem of adding hotlinks and coding theory. One can think of the index tree as the coding of words (where in a binary tree, e.g., a left move corresponds to "0" and a right move to "1"). Thus, any leaf is a code word in the code alphabet. By adding a hotlink, we actually add another letter to the alphabet. Consequently, Shannon's theorem suggests a lower bound for the problem. In particular, in binary trees, denoting by $H(p)$ the entropy of the access distribution on the leaves and denoting by T^A the hotlink-enhanced index structure resulting from the original tree T with the hotlinks added by the algorithm A,

$$ExpectedCost[T^A, p] \geq \frac{1}{\log 3} \cdot H(p) = \frac{1}{\log 3} \sum_{i=1}^{|Leaves(T)|} p_i \log\left(\frac{1}{p_i}\right),$$

and in trees of maximal degree Δ,

$$ExpectedCost[T^A, p] \geq \frac{H(p)}{\log \Delta + 1}.$$

These bounds also hold for the greedy user model.

An approximation algorithm for adding hotlinks to bounded degree trees for the greedy user model was presented in Kranakis et al. [2004]. The approximation ratio of the algorithm depends on Δ, the maximum degree of the tree, and on the entropy of the access distribution (and is in general at least $\log(\Delta + 1)$). In Czyzowicz et al. [2003] the authors propose a greedy heuristic which was tested with both simulated and real instances, where the latter were extracted from the websites of a Canadian university. For websites with simulated access patterns they report a gain of 35%, whereas for their real instance, the gain was 27%. Experimental work is also presented in Pessoa et al. [2004].

Matichin and Peleg propose a polynomial-time algorithm for approximating the gain in expected cost resulting from the selection of hotlinks in the clairvoyant user model [Matichin and Peleg 2003]. This algorithm uses greedy choices at each iteration, and achieves an approximation ratio of 2. Bose et al. [2002]

discuss the use of hotlink assignments in asymmetric communication protocols to achieve better performance bounds.

1.4 Our Results

We present an algorithm for optimally solving the hotlink enhancement problem on binary trees in the greedy user model. The algorithm uses dynamic programming and the greedy assumption to limit the search operations. We also show how to generalize the solution to arbitrary degree trees and to hotlink enhancement schemes that allow up to K hotlinks per node. In contrast with the approximation algorithm of Kranakis et al. [2004] which is polynomial for trees of bounded degree but arbitrary depth, our exact algorithm can be used for trees with unbounded degree. For an input tree T with n nodes, our algorithm runs in $O(n \cdot Depth(T)^3 \cdot 2^{2Depth(T)})$ time, requiring $O(n \cdot 2^{Depth(T)})$ space. Thus, it runs in polytime for trees of logarithmic depth.

Although the computational complexity of our algorithm may seem restrictive at first glance, in reality it is not because websites are usually designed so that a user does not need to traverse many links to reach the desired information. Moreover, a category in an index is introduced usually only when it contains more than one item (i.e., child). Hence, the height of the underlying navigational tree is small. We report experiments using data extracted from a set of selected websites. They show that our algorithm can find, in a reasonable amount of time, a set of hotlinks that provides good gain in terms of the expected number of links traversed by the user. Our experiments also give strong evidence that the greedy algorithm proposed in Czyzowicz et al. [2003] produces solutions close to the optimal. In that paper, this conclusion could not be achieved because no optimal algorithm was known.

For the case where the probability distribution on the leaves is unknown, we prove that there exist trees for which the expected tour length of the optimal enhanced structure is $\Omega(\log n)$ and then show that for any input tree T, the expected tour length of the optimal enhanced structure is $O(\log n)$. Finally, we give an exact polynomial-time algorithm for finding the best possible hotlink assignment. In contrast to the case where the probabilities are known, this algorithm is polynomial for trees with unrestricted depth.

The NP-hardness proof of Bose et al. [2000] for the hotlink enhancement problem on DAGs in the clairvoyant user model can be easily augmented to prove also the NP-hardness of the problem in the greedy model. This is true since the reduction creates a graph which has a low-cost assignment only if the original instance of the problem had a cover, but the specified assignment value is the same in both models (since there are no crossing edges), thus the same assignment value exists also in the greedy model. On the other hand, if no clairvoyant assignment achieves that value, then, necessarily, no greedy assignment can reach it either, thus the same reduction can be used with the greedy model.

The remainder of this article is structured as follows. Section 2 presents our algorithm for finding an optimal set of hotlinks and its analysis. Section 3 discusses the particular case where the frequencies of visiting each page are

unknown. In Section 4, we report our experimental results. Finally, Section 5 concludes the work.

2. KNOWN DISTRIBUTION MODEL

2.1 Basic Properties

Throughout this text, we say that a pair of hotlinks *cross* if either they cross (in the natural sense) or they share a common endpoint. We say that a set of hotlinks S is *well formed* if it does not contain any pair of crossing hotlinks (e.g., on a tree that contains edges $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6)\}$ the following set of hotlinks is well formed, and does not contain crossing hotlinks: $\{(v_1, v_6), (v_2, v_5)\}$).

Observe that for a well-formed set of hotlinks S, the greedy path from the root to any leaf v coincides with the shortest path in the hotlink-enhanced index structure. This means that the costs of a well-formed S in both the greedy and clairvoyant user models are the same.

LEMMA 2.1. *For every tree T, there exists an optimal solution to the hotlink enhancement problem in the greedy user model which is well formed.*

PROOF. The lemma is proved by arguing that if two hotlinks cross each other, then it is possible to remove the second hotlink. Assume the contrary. Then, there exist nodes v_1, v_2, v_3, v_4 such that each v_{i+1} is a proper descendant of v_i, $i = 1, 2, 3$. Moreover, v_1 and v_2 are each the origin of hotlinks ending in v_3 and v_4, correspondingly. Since destination v_4 is a descendant of v_1 and v_3, any user looking for information in v_4 or in one of its offsprings will take the first hotlink by the greedy assumption. Hence, the second hotlink will never be taken and can be removed without increasing the path length $|D_S|$ of any of the nodes. □

If two hotlinks have the same destination then, by definition, they cross. Thus we have the following corollary.

COROLLARY 2.2. *There exists an optimal solution to the hotlink enhancement problem in which no two hotlinks arrive at the same node.*

The usefulness of these results stems from the fact that they help to narrow down the domain of potential solutions that needs to be searched (in the greedy user model) in order to find the optimal one.

We first restrict our discussion to n-node binary trees and to the case where $K = 1$. We represent a solution S as a function $S : V \mapsto V$, with $S(v)$ being the endpoint of the hotlink starting at v. We note that our algorithms may end up with a solution S where the value of S is not determined for some subset of nodes, which means that there is no hotlink leaving nodes in such a set.

For every node v in the tree T, let T_v denote the subtree of T consisting of v and all its descendants, and let $\mathcal{P}_v$ denote the path leading from the root to v in T (in addition, $\mathcal{P}_v$ includes v itself).

We generalize the definition of S to a function $S : V \mapsto 2^V$. For a node v, the set $S(v)$ is referred to as an *undetermined hotlink*, and interpreted as

the collection of candidates to be the final endpoint of the hotlink from v. An undetermined hotlink function S is said to be *determined* or *fixed* for a node set $W \subseteq V$ if $S(w)$ is a singleton for every $w \in W$.

The cost associated with the solution S on the subtree T_v, assuming S is determined for $\mathcal{P}_v \cup T_v$, is

$$c(v, S) = \sum_{w \in Leaves(T_v)} |D_S(w)| \cdot p(w) .$$

The cost of S over the entire tree T with root r is thus $f(T, p, S) = c(r, S)$.

An undetermined hotlink function S is said to be *separated with respect to* the node v (or simply *v-separated*) if the nodes of $\mathcal{P}_v$ are partitioned into three disjoint sets, $\mathcal{P}_v = \mathcal{P}_v^F(S) \cup \mathcal{P}_v^I(S) \cup \mathcal{P}_v^O(S)$, called the *fixed-set*, *in-set* and *out-set*, respectively, such that:

(1) For every $w \in \mathcal{P}_v^F(S)$, $S(w)$ is a single proper descendant of w in $\mathcal{P}_v$;
(2) for every $w \in \mathcal{P}_v^I(S)$, $S(w) \subseteq T_v \setminus \{v\}$; and
(3) for every $w \in \mathcal{P}_v^O(S)$, $S(w) \subseteq V \setminus (\mathcal{P}_v \cup T_v)$.

We remark that our algorithm will consider candidate solutions separated with respect to v in which for every node w in the in-set we have equality, namely, $S(w) = T_v \setminus \{v\}$, and similarly, for every node w in the out-set we have $S(w) = V \setminus (\mathcal{P}_v \cup T_v)$.

For two solutions S_1 and S_2 and a node set W, we say that S_1 *is compatible with S_2 on W* if $S_1(v) \subseteq S_2(v)$ for every node $v \in W$. Moreover, S_1 is compatible with S_2 if it is compatible with S_2 on the entire node set of T.

Note that for an undetermined hotlink function S, the greedy path $D_S(w)$ is not necessarily defined for every leaf w of T. For this path to be defined uniquely, it is necessary that $D_{S'}(w)$ remains the same for every determined function S' compatible with S. In this case, we say that S is *route-determined* for w. This means in particular that the cost of accessing w is determined.

The next observation follows from the previous definitions.

Observation 2.3. (a) If a hotlink function S is determined for $\mathcal{P}_v \cup T_v$, then it is route-determined for every leaf w in T_v. (b) If a v-separated hotlink function S is determined for $\mathcal{P}_v^I \cup T_v$, then it is route-determined for every leaf w in T_v.

Observation 2.3 facilitates the use of a dynamic programming approach on the problem, as it leads to the observation that the cost of any solution S over the subtree T_v is determined solely on the basis of its values on $\mathcal{P}_v \cup T_v$, and more importantly, the cost of any v-separated solution S over the subtree T_v is determined solely on the basis of its values on $\mathcal{P}_v^F \cup \mathcal{P}_v^I \cup T_v$. More formally, we have the next lemma.

LEMMA 2.4. *Consider two hotlink functions S_1 and S_2 that are both v-separated with the same fixed-set $\mathcal{P}_v^F(S_1) = \mathcal{P}_v^F(S_2)$, and the same in-set $\mathcal{P}_v^I(S_1) = \mathcal{P}_v^I(S_2)$. If S_1 and S_2 are determined in the same way on $\mathcal{P}_v^F \cup \mathcal{P}_v^I \cup T_v$, that is, if $S_1(w) = S_2(w)$ for every $w \in \mathcal{P}_v^F \cup \mathcal{P}_v^I \cup T_v$, then $c(v, S_1) = c(v, S_2)$.*

2.2 A First Algorithm

Here, we present an algorithm for the hotlink enhancement problem. An optimized version of this algorithm is presented in Section 2.3.

The recursive procedure Proc employed by the algorithm receives as its input a node v in the tree, and a v-separated partial hotlink function S of a specific form. Suppose that v is of depth d from the root, and let $\mathcal{P}_v = (root = v_0, v_1, \ldots, v_d = v)$. Then, S will be specified as a vector $\bar{s} = \langle s(0), \ldots, s(d-1)\rangle$, where

$$s(i) = \begin{cases} j, & \text{if } S(v_i) = v_j \text{ for } i+2 \le j \le d, \\ I, & \text{if } S(v_i) = T_v \setminus \{v\}, \\ O, & \text{if } S(v_i) = V \setminus (\mathcal{P}_v \cup T_v). \end{cases}$$

The goal of procedure Proc$(v, \bar{s})$ is to calculate the cost of the optimal completion of S on $\mathcal{P}_v^I(S) \cup T_v$. In particular, to find the optimal cost $\min_S\{f(T, p, S)\}$ of the hotlink enhancement problem, one needs to run Proc$(r, \bar{0})$, where $\bar{0}$ is an empty vector and r is the root of the tree (there are no nodes in $\mathcal{P}_r$).

Procedure Proc operates as follows:

(1) If T_v is small enough (e.g., contains fewer than K_0 nodes for some constant K_0), then the optimal cost is found by exhaustive search, examining all possible completions.

(2) Now suppose the tree T_v is larger than K_0 nodes. Denote the children of v by v_L and v_R. We aim to generate all possible v_L and v_R separated partial hotlink functions that are compatible with S. The procedure examines all the possible ways of partitioning the set $\mathcal{P}_v^I$ (including v itself) into four disjoint sets named H_L, H_R, B_L, and, B_R. Set H_L will be interpreted as the set of nodes that have a hotlink directed to v_L (in any solution it is enough to have only one such hotlink). If $H_L = \emptyset$, then in the current completion there is no hotlink to be ended directly in node v_L. Set B_L will be interpreted as the collection of start points of hotlinks to be ended at the nodes of T_{v_L}, except v_L itself (the left-side subtree). The sets B_R and H_R are defined analogously for the righthand side of the tree T_v. At the end of this step we get all v_R-separated and v_L-separated hotlink functions possible from S.

(3) For each such partition (H_L, H_R, B_L, B_R) constructed from S, the procedure acts as follows.

(3.1) It generates the (partial) solution S_L derived from S by specifying that the hotlink starting from the node in H_L (if such exists) ends at v_L, the hotlinks from the nodes of B_L (if such exist) end inside T_{v_L}, and the hotlinks from the nodes of $H_R \cup B_R$ end outside $\mathcal{P}_{v_L} \cup T_{v_L}$. In other words, the vector representation of the generated S_L is

$$s_L(i) = \begin{cases} d+1, & i \in H_L, \\ O, & i \in B_R \cup H_R, \\ s(i), & \text{otherwise.} \end{cases}$$

(Note that in particular, $s_L(i)$ remains I for nodes v_i of B_L, and maintains its previous value (which is either O or some integer $i+2 \le j \le d$) for nodes outside $\mathcal{P}_v^I$).

(3.2) It similarly generates the partial solution S_R derived from S by following the specifications of the partition for the right subtree T_{v_R}.

(4) For each generated pair (S_L, S_R) of partial functions, the procedure proceeds as follows. First, it tests whether either S_R or S_L defines crossing hotlinks. (In fact, we generate only those sets that do not include crossings; that is, if $\mathcal{P}_v^I = \{u_1, u_2, \ldots u_i, u_{i+1}, \ldots\}$ and u_i is selected to H_L, then generate only B_L's that do not contain $u_{i+1}, u_{i+2}, \ldots$). Then, if no crossing hotlinks are defined, the procedure is invoked recursively on $(v_L, \bar{s}_L)$ and $(v_R, \bar{s}_R)$, and returns cost values x_L and x_R, respectively.

(5) The procedure returns the lowest combined cost $x_L + x_R$ among all pairs of examined functions.

Note that the algorithm invokes the procedure by dynamic programming, rather than plain recursion. Specifically, it maintains a table of costs $A(v, \bar{s})$ for every node v and partial solution $\bar{s}$ of the type described earlier. Whenever the procedure requires an answer for some pair $(v, \bar{s})$, it first consults table A. Hence, each entry in the table must be computed only once, namely, on the first occasion that it is requested.

LEMMA 2.5. *For every node v in T and for every v-separated hotlink function S, procedure* Proc *returns the cost of the determination for $(v, \bar{s})$ with minimal cost.*

PROOF. The proof is by structural induction on the tree T, from the leaves up. For the induction basis, we consider the case where $|T_v| < K_0$, where K_0 is the constant of step 1. In this case, the result holds due to the exhaustive search performed in this step.

Let us now consider the case where $|T_v| \geq K_0$. Let S^* the determined hotlink function with minimum cost among those that are compatible with S. Also, let S_L^* be a v_L-separated function defined as follows.

$$S_L^*(u) = \begin{cases} S^*(u), & \text{if } S^*(u) \in P_{v_L}, \\ T_{v_L} \setminus \{v_L\}, & \text{if } S^*(u) \in T_{v_L} \setminus \{v_L\}, \\ V \setminus (P_{v_L} \cup T_{v_L}), & \text{if } S^*(u) \in V \setminus (P_{v_L} \cup T_{v_L}) \\ S(u) & \text{if there is no hotlink leaving } u \text{ in } S^* \end{cases}$$

The function S_R^* is defined in a similar way. By construction, both S_L^* and S_R^* are compatible with S. In addition, note that the cost of S^* is the cost of the optimal completion for S_L^* added to the cost of the optimal completion for S_R^*, for otherwise there would exist a completion for S with cost smaller than that of S^*. Since Proc considers all possibilities to generate pairs of v_L- and v_R-separated hotlink functions that are compatible with S, then, in particular, Proc considers the pair (S_R^*, S_L^*). It follows from induction that Proc calculates the best completion for both S_R^* and S_L^*. Thus, the cost returned by Proc when executed for $(v, \bar{s})$ is the cost of the optimal completion for S_R^* added to the cost of the optimal completion for S_L^*, which turns out to be the cost of S^* □

2.3 Refined Algorithm

Here, we present a refined version of Proc. The two algorithms are similar in the order of their time complexities, though the refined one checks only a subset of the cases checked by the previous version. However, there are several reasons for introducing the refined version. First, it turned out easier to use in our experiments (moreover, in the practical version, saving even by a constant in the time complexity can be very helpful). Second, it saves also in terms of memory needed (the size of the table); this, too, helps us in using the algorithm in practice. Finally, we needed the compact representation defined here as a basis for the design and the implementation of an algorithm for unknown probabilities (introduced in the next section).

The key idea for this new procedure is the concept of signatures. Before introducing it, let us consider an illustrative example.

Example 2.6. Let $\mathcal{P}_v = (root = v_0 v_1 \cdots v_8 = v)$ be a path on the input tree T and let S_1 and S_2 be two v-separated hotlink functions with representations

$$\overline{s}_1 =< 4, 3, O, O, I, I, 8, O >$$

and

$$\overline{s}_2 =< 5, O, O, O, O, I, I, O >,$$

respectively. Careful analysis shows that Proc fills the same value for the entries corresponding to (S_1, v) and (S_2, v) in the dynamic programming table. To see this, let α and β, with $\alpha \leq \beta$, be the values of the entries (S_1, v) and (S_2, v), respectively. Let S_1^* be a determined function compatible with S_1 such that $c(v, S_1^*) = \alpha$. On the other hand, let S_2^* be a determined function compatible with S_2 constructed in such a way that $S_2^*(v_5) = S_1^*(v_4)$, $S_2^*(v_6) = S_1^*(v_5)$, and $S_2^*(u) = S_1^*(u)$, for every $u \in T_v$. By inspection, one can conclude that $c(v, S_2^*) = \alpha$. Since β is the minimum value of $c(v, S)$ among all determined functions S compatible with S_2, we get that $\beta \leq c(v, S_2^*)$. Thus $\beta = \alpha$.

The preceding example illustrates the fact that, under certain conditions, one can deduce that two distinct v-separated functions yield the same value even without evaluating them. It should be clear that this kind of deduction can be useful for reducing the size of the dynamic programming table. Thus the following question arises: *What conditions do guarantee that two distinct v-separated functions yield to the same value?* The signature concept presented to follow is a response (at least a partial one) for such a question since, as we prove, if two v-separated functions share the same signature, then their entries in the dynamic programming table are associated with the same value. The following definition generalizes the notion of the greedy path to the case of an undetermined function.

Definition 2.7 (*The Greedy Path of an Undetermined Function*). Let S be a v-separated function. The greedy path $D_S(v)$ followed by the user when searching node v is the greedy path followed by the user to find v in a DAG, obtained from the input tree T by the addition of hotlinks in the set $\{(u, S(u)) | u \in \mathcal{P}_v^F\}$.

Definition 2.8 (*Signature*). Let S be a v-separated function and let $D_S(v) = (root = u_0, u_1, \ldots, u_l = v)$ be the greedy path followed by the user when searching node v. We say that a node $u \in D_S(v)$ is available if and only if $u \in \mathcal{P}_v^I(S)$. The signature of S is a bit vector of size $|D_S(v)|$, where the bit associated with u_i is 1 if and only if u_i is available.

In Example 2.6, $D_{S_1}(v) = v_0v_4v_5v_6v_8 = v$ and $D_{S_2}(v) = v_0v_5v_6v_7v_8 = v$. Furthermore, both S_1 and S_2 have signature $(0, 1, 1, 0, 1)$. (Note that node v is always available).

LEMMA 2.9. *Let S_1 and S_2 be two v-separated functions that have the same signature. Furthermore, let S_1^* be a determined function compatible with S_1. Then:*

(i) *There is a determined function S_2^*, compatible with S_2, such that $c(v, S_2^*) = c(v, S_1^*)$.*
(ii) Proc *fills the same value for the entries (S_1, v) and (S_2, v).*

PROOF. First we prove (i). We construct a function S_2^* compatible with S_2 in such a way that both of the following properties hold.

(1) Let t be the number of available nodes in $D_{S_1}(v)$. Then, for $i = 1, \ldots, t$, the value of S_2^* on the ith available node of $D_{S_2}(v)$ is equal to that of S_1^* on the ith available node of $D_{S_1}(v)$;
(2) $S_2^*(w) = S_1^*(w)$, for every $w \in T_v$.

Note that the aforesaid construction is possible because $|D_{S_1}(v)| = |D_{S_2}(v)|$ and the number of available nodes in $D_{S_1}(v)$ is equal to that of $D_{S_2}(v)$.

Let us now show that $c(v, S_2^*) = c(v, S_1^*)$ by arguing that $|D_{S_1^*}(L)| = |D_{S_2^*}(L)|$, for every $L \in Leaves(T_v)$. Given such an L, let u^1 be the first node of $D_{S_1}(v)$ such that $S_1^*(u^1) \in T_v$ and $S_1^*(u^1)$ is an ancestor of L in T. Similarly, let u^2 be the first node of $D_{S_2}(v)$ such that $S_2^*(u^2) \in T_v$ and $S_2^*(u^2)$ is an ancestor of L in T. The definition of S_2^* and the fact that S_1 and S_2 have the same signature ensure that u^1 is the ith node (starting from the root) in the path $D_{S_1^*}(L)$ if and only u^2 is the ith node (starting from the root) in the path $D_{S_2^*}(L)$. Therefore, the subpath of $D_{S_1^*}$ that starts at *root* and ends at $S_1^*(u^1)$ has the same size of that of $D_{S_2^*}$ that starts at *root* and ends at $S_2^*(u^2)$

On the other hand, the subpath of $D_{S_2^*}(L)$ that starts at $S_2^*(u^2)$ and ends at L is equal to that of $D_{S_1^*}(L)$ that starts at $S_1^*(u^1)$ and ends at L. This is because $S_2^*(u^2) = S_1^*(u^1)$ and $S_2^*(w) = S_1^*(w)$, for every $w \in T_v$.

The previous discussion allows us to conclude that $|D_{S_1^*}(L)| = |D_{S_2^*}(L)|$, which establishes (i).

For the proof of (ii), let α and β be the values of the entries (S_1, v) and (S_2, v), respectively. We assume without loss of generality that $\alpha \leq \beta$. Furthermore, let S_1^* be a function compatible with S_1 such that $c(v, S_1^*) = \alpha$ and let S_2^* be the function compatible with S_2 such that $c(v, S_2^*) = \alpha$. Since β is the minimum value of $c(v, S)$ among all determined functions S compatible with S_2, we get that $\beta \leq c(v, S_2^*) = \alpha$ and, as a consequence, $\beta = \alpha$. □

The previous result allows us to reduce the number of recursive calls made by Proc. Before calling Proc for (S, v) recursively, we add the following test. We now check whether the cost of optimal completions for the v-separated functions whose signatures coincide with that of S has already been calculated. By the previous lemma, all of these completions have the same cost. Hence, if the desired value has already been calculated, then the revised Proc does not call Proc(S, v).

In order to implement the aforementioned test, our refined procedure maintains a dynamic programming table A with one entry for each pair in the set $V \times \mathcal{S}g$, where V is the set of nodes of T and $\mathcal{S}g$ is the set of binary strings of size at most $Depth(T)$. The entry A$[v, g]$ stores the cost achieved by the optimal completions of v-separated functions with signature g. For the sake of completeness and further reference, we present a detailed description of the refined procedure to follow. Note that apart from introducing table A, the only difference with respect to Proc is in step 4.

(1) If T_v is small enough (e.g., contains fewer than K_0 nodes, for some constant K_0), then the optimal cost is found by exhaustive search, examining all possible completions.

(2) Now, suppose the tree T_v is larger than K_0 nodes. Denote the children of v by v_L and v_R. We aim to generate all possible v_L- and v_R-separated partial hotlink functions that are compatible with S. The procedure examines all possible ways of partitioning set $\mathcal{P}_v^I$ (including v itself) into four disjoint sets named H_L, H_R, B_L, and B_R. Set H_L will be interpreted as the set of nodes that have a hotlink directed to v_L (in any solution it is enough to have only one such hotlink). If $H_L = \emptyset$, then in the current completion there is no hotlink to be ended directly in node v_L. The set B_L will be interpreted as the collection of start points of hotlinks to be ended at the nodes of T_{v_L} except v_L itself (the left-side subtree). Sets B_R and H_R are defined analogously for the righthand side of the tree T_v. At the end of this step we get all v_R-separated and v_L-separated hotlinks functions possible from S.

(3) For each such partition (H_L, H_R, B_L, B_R) constructed from S, the procedure does the following.

  (3.1) It generates the (partial) solution S_L, derived from S by specifying that the hotlink from the node of H_L (if such exists) ends at v_L, the hotlinks from the nodes of B_L (if such exist) end inside T_{v_L}, and those from the nodes of $H_R \cup B_R$ end outside $\mathcal{P}_{v_L} \cup T_{v_L}$. In other words, the vector representation of the generated S_L is

$$s_L(i) = \begin{cases} d+1, & i \in H_L, \\ O, & i \in B_R \cup H_R, \\ s(i), & \text{otherwise.} \end{cases}$$

  (Note that in particular, $s_L(i)$ remains I for nodes v_i of B_L, and maintains its previous value, which is either O or some integer $i+2 \le j \le d$, for nodes outside $\mathcal{P}_v^I$).

  (3.2) It similarly generates the partial solution S_R, derived from S by following the specifications of the partition for the right subtree T_{v_R}.

(4) For each generated pair (S_L, S_R) of partial functions, the procedure acts as follows. First, it tests whether either S_R or S_L defines crossing hotlinks. Then, if no crossing hotlinks are defined, it retrieves the entries $A[v_L, g_L]$ and $A[v_R, g_R]$ in the dynamic programming table, where g_L and g_R are the signatures of S_L and S_R, respectively. Furthermore, it executes the following steps.
   (4.1) If the entry $A[v_L, g_L]$ is not empty, then $x_L \leftarrow A[v_L, g_L]$.
   (4.2) Otherwise, if the entry $A[v_L, g_L]$ is empty, the procedure is invoked recursively on $(v_L, \bar{s}_L)$. The returned cost x_L is stored in $A[v_L, g_L]$.
   (4.3) If the entry $A[v_R, g_R]$ is not empty, then $x_R \leftarrow A[v_R, g_R]$.
   (4.4) Otherwise, if the entry $A[v_R, g_R]$ is empty, the procedure is invoked recursively on $(v_R, \bar{s}_R)$. The returned cost x_R is stored in $A[v_R, g_R]$.

(5) The procedure returns the lowest combined cost $x_L + x_R$ among all pair of functions examined.

We can state the following lemma.

LEMMA 2.10. *For every node v in T, and for every v-separated hotlink function S, the refined* Proc *returns the cost of the determination for $(v, \bar{s})$ with minimum cost.*

PROOF. It follows from both Lemmas 2.5 and 2.9. □

After filling the table A, the optimal set of hotlinks can be obtained by calling the following procedure with $v = r$, the root of the input tree T, and $\bar{s} = \bar{0}$, where $\bar{0}$ is the empty vector.

Algorithm. Find-Solution $(v, \bar{s})$

---

1. $MIN \leftarrow +\infty$
2. For each pair of functions (S_L, S_R) generated from the partitions of S, as in the steps 2–3 of Proc, the algorithm tests if either S_R or S_L defines crossing links. If none of them does then the steps presented next are executed.
   2.1. Let g_L and g_R be the signatures of S_L and S_R, respectively.
   2.2. If $MIN > A[v_L, g_L] + A[v_R, g_R]$ then MIN is updated to $A[v_L, g_L] + A[v_R, g_R]$ and (S_L, S_R) is stored as the best pair of functions found so far.
3. Let (S_L^*, S_R^*) be the best pair of functions found. If S_L^* defines a hotlink from a node u_L to v_L, then (u_L, v_L) is added to the optimal set of hotlinks. Similarly, if S_R^* defines a hotlink from a node u_R to v_R, then (u_R, v_R) is added to the optimal set of hotlinks.
4. Run the procedure recursively for $(v_L, \bar{s^*}_L)$ and for $(\bar{s^*}_L, v_R)$.

2.3.1 *Complexity Analysis.* We note that the refined Proc is an optimized version of Proc. Nevertheless, let us now analyze its time and space complexities.

For the space complexity, we note that the dynamic programming table has size $O(n2^{Depth(T)})$, since for every node v, the maximum number of v-separated functions with distinct signatures is $O(2^{Depth(T)})$ and we have at most n possibilities for v.

In order to determine the time complexity, we note that each invocation requires us to classify all the possible partitions of a set of size at most $Depth(T)$

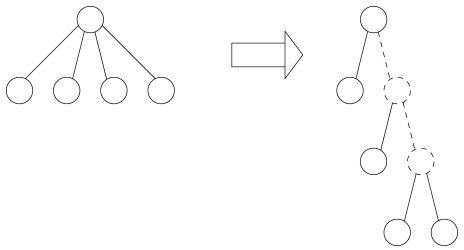

Fig. 2. The transformation from degree $\Delta = 4$ to degree 2.

into four subsets (two of which are at most singletons). Furthermore, for each partition we have to generate the corresponding partial hotlink functions and calculate their signatures. Since all these computations can be done in $O(Depth(T))$ time and the number of partitions is $O(Depth(T)^2 \cdot 2^{Depth(T)})$, it follows that our algorithm runs in $O(n \cdot Depth(T)^3 \cdot 2^{2Depth(T)})$ time, which is polynomial in n when $Depth(T) = O(\log n)$.

THEOREM 2.11. *There exists a polynomial-time algorithm for solving the hotlink enhancement problem with known probability distribution on n-node binary trees of depth* $O(\log n)$.

2.4 Handling Nonbinary Trees and Information in Nonleaf Nodes

The solution as described earlier applies only to binary trees and assumes that the sought information items may be stored only in the leaves. Let us next outline the way to generalize it so that it applies to trees of arbitrary degree and to situations when information may be stored also in nonleaf nodes.

First, let us generalize the definition of the problem in three ways.

—Associate with each edge e of the tree T a *length* $\ell(e)$. The length of a root-to-leaf path is now measured as the sum of the edge lengths along the path, rather than the number of edges. Assign some constant-length value to hotlinks.

—Certain nodes can be tagged as "no-exit" nodes; it is not allowed to start a hotlink at such a no-exit node.

—Certain nodes can be tagged as "no-entry" nodes; it is not allowed to end a hotlink at a no-entry node.

It is not hard to show that the algorithm presented earlier in this section can be modified to handle the generalized problem as well.

Given the modified algorithm, we now show how any instance (T, p, ℓ) of the problem, where T is a tree of maximum degree Δ, can be transformed into an instance (T', p', ℓ'), where T' is a binary tree, such that $f(T, p, \ell, S) = f(T', p', \ell', S)$ for every hotlink function S. This is done by replacing each node v of degree $d_v > 2$ in T with a chain of $d_v - 2$ new nodes, splitting the load of v along this chain. The new nodes are designated "no-exit, no-entry", and the new edges assigned zero length. The weights p of the leaves and the length of all old edges remain the same as in T (see Figure 2).

LEMMA 2.12. *If* (T', p', ℓ') *is obtained from* (T, p, ℓ) *by the aforesaid transformation, then* $f(T, p, \ell, S) = f(T', p', \ell', S)$ *for every hotlink function* S.

It follows that $\hat{f}(T, p, \ell) = \hat{f}(T', p', \ell')$. Also, observe that while the depth of the tree T' may now be greater than that of T, at most $Depth(T)$ nodes along any root-to-leaf path in T' are *not* marked "no entry, no exit", and those nodes that are marked "no entry, no exit" do not influence the number of possible signatures. Hence we have the following corollary.

COROLLARY 2.13. *There exists a polynomial-time algorithm for solving the hotlink enhancement problem with known probability distribution on n-node trees of depth* $O(\log n)$.

Information stored in nonleaf nodes is handled in a similar manner. Suppose that a nonleaf node v stores some information items. Create a new leaf v', attach it to v, designate v' a no-entry node and assign the edge from v to v' zero length. Again, one can readily verify the following.

LEMMA 2.14. *If* (T', p', ℓ') *is obtained from* (T, p, ℓ) *by the aforesaid transformation, then* $f(T, p, \ell, S) = f(T', p', \ell', S)$ *for every hotlink function* S.

COROLLARY 2.15. *There exists a polynomial-time algorithm for solving the hotlink enhancement problem with known probability distribution on n-node trees of depth* $O(\log n)$ *where information may be stored in every node.*

2.5 Handling More than One Hotlink

Consider the case $K > 1$, namely, where more than one hotlink is allowed at each node. In this case, we can expand each node v into a long chain of K nodes. The length of the edges between them is set to zero ($\ell(e) = 0$) and all of these K nodes are labeled no-entry except for the first. This way, we can associate all the hotlinks from these nodes (where each node is allowed only one hotlink) with the hotlinks of v. It is easy to verify that by searching for a solution to the new modified tree, we obtain a solution for the new problem. The depth of the new tree is $Depth(T) \cdot K$, thus for $K = O(1)$ it is still $O(Depth(T))$.

COROLLARY 2.16. *There exists a polynomial-time algorithm for solving the hotlink enhancement problem with known probability distribution on n-node trees of depth* $O(\log n)$ *where every node can have up to* $K = O(1)$ *hotlinks directed from it and information may be stored in every node.*

3. UNKNOWN DISTRIBUTION

In this section, we consider the case where the probabilities associated with the leaves are not known in advance.

Most of the results in this section employ the equivalence, formalized in Corollary 1.2, between the problem of finding the set of hotlinks which minimizes the expected access cost when the distribution probability is unknown, and that of finding the set of hotlinks which minimizes the longest greedy path from the root to a leaf in the enhanced structure.

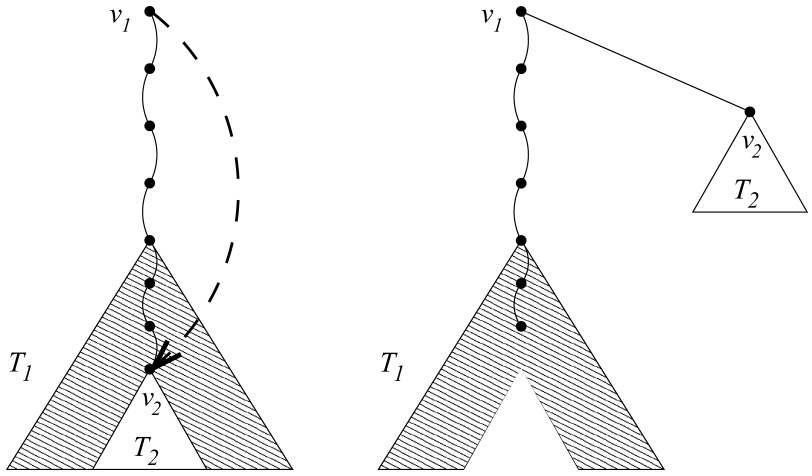

Fig. 3. The hotlink elimination step.

3.1 Lower Bounds and Upper Bounds

Here, we prove both lower bounds and upper bounds on the expected access cost under an unknown probability distribution on the leaves. The arguments employed here are similar to those employed in Kranakis et al. [2004] for the case where the probability distribution on the leaves is known.

Throughout this section, we assume again that sought information items are stored only in leaves. Let m be the number of leaves in T. First of all, we prove a lower bound on the expected access cost under an unknown probability distribution on the leaves.

We assume that tree T is Δ-ry, that is, its degree is at most Δ for some constant $\Delta \geq 1$, and that K hotlinks are allowed from each node.

Observe that if there is a hotlink leading to some node w in T, then the tree edge leading to w from its parent in T will never be used in a greedy route to any leaf of T_w. This observation implies that for any set of hotlinks S, the solution resulting from adding S to T is equivalent in cost to some $(\Delta + K)$-ry tree T' with no hotlinks at all. Tree T' can be obtained from the pair (T, S) by performing the following modification: For each hotlink leading from v to w, eliminate the edge connecting w to its parent from the tree, and replace the hotlink by a tree edge connecting v to w (see Figure 3). Hence, the cost of any solution using up to K hotlinks on a Δ-ry tree is bounded from below by the cost of the best solution using no hotlinks on a $(\Delta + K)$-ry tree.

For an integer $\ell \geq 1$, the number of distinct leaves reachable from the root in ℓ steps on a $(\Delta + K)$-ry tree is bounded from above by $(\Delta + K)^\ell$. This implies that a solution S in which each of the m leaves is reachable by a path of length ℓ or less, that is, with $D_S(v) \leq \ell$, must satisfy $(\Delta + K)^\ell \geq m$, or $\ell \geq \frac{\log m}{\log(\Delta+K)}$. Hence for constant Δ and K, $\ell = \Omega(\log m)$.

Using Corollary 1.2 we get the following.

COROLLARY 3.1. *For any m-leaf Δ-ry tree T, if at most K hotlinks are allowed from each node, then* $\tilde{f}(T) = \frac{\log m}{\log(\Delta+K)} - 1$. *In particular, for constant* Δ *and* K, $\tilde{f}(T) = \Omega(\log m)$.

Let us now show how to construct a generic set of hotlinks that guarantees a global upper bound of $O(\log n)$ on the access cost, which is asymptotically tight whenever $m = \Theta(n)$. In fact, this can be achieved using a single hotlink per node. This result is for the worst case. However, later in this section it is used for developing and analyzing a polynomial-time algorithm that is optimal for every input tree (of any depth). We rely on the following well-known fact.

LEMMA 3.2. *Given an n-node tree T, it is always possible to find a separating node v whose removal breaks the tree into subtrees of size at most* $n/2$.

Using this lemma, we define the hotlinks for our tree T rooted at r as follows. Let $w_1, w_2, \ldots, w_k$ be r's children in T and let $T - T_v$ be the tree obtained by removing the nodes of T_v from T. First, find a separator node v for T as in the lemma, and add a hotlink from r to v. Let us assume without loss of generality that v belongs to T_{w_k}. Next, generate hotlinks for $T_{w_1}, T_{w_2}, \ldots, T_{w_{k-1}}, T_{w_k} - T_v$, and T_v by applying the same procedure recursively.

Let $f(n)$ denote the maximum root-to-leaf distance in an n-node tree with hotlinks generated by the previous procedure. Note that the construction guarantees that $f(n) \leq 2 + f(n/2)$. By solving this recurrence, we conclude that $f(n) \leq 2\log n + 1$.

It follows that no matter what the probabilities are, the resulting cost using this construction is always $O(\log n)$.

LEMMA 3.3. *For any n-node tree T, the algorithm described earlier constructs a hotlink function S such that* $\tilde{f}(T, S) \leq 2\log n + 1$. *Moreover, every node in the resulting enhanced structure (not just the leaves) is at distance at most* $2\log n + 1$ *from the root.*

Given any optimal solution, it cannot be worse than those obtained by Lemma 3.3. Hence, in any optimal enhanced structure for a tree T, every leaf of T is located at depth $O(\log n)$ (in the enhanced structure of T). However, one can envision optimal enhanced structures in which the internal nodes of T are located much further than that. (For example, consider the case that T is just a line of n nodes of degree two each, except for the root and the single leaf, which are of degree one each; this optimal enhanced structure contains only one hotlink from the root to the single leaf. Note that many internal nodes of T are located at distance $\Omega(n)$).

We develop a polynomial-time algorithm for trees of any depth for the unknown probabilities case. The construction and analysis of that algorithm rely on the existence of a logarithmic upper bound on the depth of all the nodes of T, not only the leaves, in an optimal enhanced structure. For this, we use Lemma 3.3 to convert any given optimal enhanced structure to be of logarithmic depth. Define $\tilde{H}$ as

$$\tilde{H}(T, S) = \max_{v \in Nodes(T)} \{|D_S(v)|\}.$$

LEMMA 3.4. *For any n-node tree T, there exists a set of hotlinks S' such that $\tilde{f}(T, S') = \tilde{f}(T)$ and $\tilde{H}(T, S') \leq 4 \log n + 2$.*

PROOF. Let S be a set of hotlinks such that $\tilde{f}(T) = \tilde{f}(T, S)$. It follows from Lemma 3.3 that $\tilde{f}(T, S) \leq 2 \log n + 1$. Let $S_<$ be the subset of hotlinks whose endpoints are located at depth at most $2 \log n + 1$ in the enhanced structure obtained from T due to the addition of the hotlinks in S.

Now, let T' be the enhanced structure obtained from T due to the addition of the hotlinks in $S_<$. Moreover, let P be the set of nodes located at depth $2\lfloor \log n \rfloor + 2$ in T' (recall that an enhanced structure can be viewed as a tree without hotlinks). Note that all the elements from P are internal nodes in T. For a node $v \in P$, let T'_v denote the subtree of T' consisting of v and all of its descendants. Let also S_v be the set of hotlinks obtained by applying the procedure described at the beginning of this section on the tree T'_v. Then define S' as

$$S' = S_< \cup \bigcup_{v \in P} S_v.$$

It is easy to verify, using Lemma 3.3, that $\tilde{H}(T, S') \leq 4 \log n + 2$. □

3.2 An Exact Polynomial-Time Algorithm

The exact algorithm presented in this section is similar to the one proposed in Section 2.3 except for two differences.

—The evaluation of the objective function in step 5 is modified; and
—we present a way to reduce the search space in the dynamic programming execution. This is how the algorithm is made to be a polynomial-time algorithm for any depth.

3.2.1 *Changing the Objective Function.* Our first modification with respect to the algorithm presented in Section 2.3 consists of replacing its fifth step by the following.

5. The procedure returns the lowest value for $\max\{x_L, x_R\}$ among all pairs of functions examined.

Let Proc^w be the algorithm obtained by this single modification. Clearly, it runs in polynomial time for trees with depth $O(\log n)$. Our second modification, explained next, allows the algorithm to run in polynomial time for trees of arbitrary depths.

3.2.2 *Discarding Large Paths.* If we execute Proc^w (without further modifications) we may generate an exponential number of subproblems for trees whose depth is not $O(\lg n)$. The following results allow us to discard many subproblems, so we end up having a polynomial number of them.

PROPOSITION 3.5. *Let S be a v-separated function and let S' be a determined function compatible with S. Then $D_S(v) = D_{S'}(v)$.*

PROOF. Recall that the greedy path $D_S(v)$ is exactly that followed by the user to find v in a DAG obtained from the input tree T due to the addition of hotlinks in the set $\{(u, S(u))|u \in \mathcal{P}_v^F\}$. Since S' is compatible with S, then $S'(v) = S(v)$ if $v \in \mathcal{P}_v^F$ and $S'(v) \notin \mathcal{P}_v$, otherwise. Thus, we can conclude that $D_{S'}(v)$ coincides with $D_S(v)$.

LEMMA 3.6. *Let S be a v-separated function whose signature has size larger than $4\log n + 2$. Then, there is an optimal hotlink function that is not compatible with S.*

PROOF. Since Lemma 3.4 ensures the existence of an optimal hotlink function S^* with $\tilde{H}(T, S^*) \leq 4\log n + 2$, it suffices to argue that $\tilde{H}(T, S') > 4\log n + 2$, for *every* determined function S' compatible with S. Let S' be a determined function compatible with S. Proposition 3.5 ensures that $D_{S'}(v) = D_S(v)$. Since $|D_S(v)| > 4\log n + 2$ we are done. □

Based on the preceding lemma we can abort the execution of step 4 (that of Section 2.3) if either g_L or g_R has size larger than $4\log n + 2$. The detailed description for the new algorithm is presented next.

(1) If T_v is small enough (e.g., contains fewer than K_0 nodes, for some constant K_0), then the best determination and its cost are found by exhaustive search, examining all possible completions.
(2) Now, suppose that tree T_v contains more than K_0 nodes. Denote the children of v by v_L and v_R. The procedure examines all possible ways of partitioning the set $\mathcal{P}_v^I \cap D_S(v)$ (including v itself) into four disjoint sets named H_L, H_R, B_L, and B_R.
(3) For each such partition (H_L, H_R, B_L, B_R) constructed from S, the procedure does the following.
    (3.1) It generates the (partial) solution S_L, derived from S by specifying that the hotlink from the node of H_L (assuming its existence) ends at v_L, the hotlinks from the nodes of B_L (assuming their existence) end inside T_{v_L}, and the hotlinks from the nodes of $H_R \cup B_R$ end outside $\mathcal{P}_{v_L} \cup T_{v_L}$. In other words, the vector representation of the generated S_L is
$$s_L(i) = \begin{cases} d+1, & i \in H_L, \\ O, & i \in B_R \cup H_R, \\ s(i), & \text{otherwise.} \end{cases}$$
    (3.2) It similarly generates the partial solution S_R, derived from S by following the specifications of the partition for the right subtree T_{v_R}.
(4) For each generated pair of partial functions S_L and S_R, the procedure proceeds as follows. First, it tests whether either S_R or S_L defines crossing hotlinks. If none of them defines crossing hotlinks, the procedure computes the signatures g_L and g_R for S_L and S_R, respectively. If neither g_L nor g_R has size larger than $4\log n + 2$, then it retrieves the entries $A[v_L, g_L]$ and $A[v_R, g_R]$ in the dynamic programming table and executes the following steps.

(4.1) If the entry $A[v_L, g_L]$ is not empty, then $x_L \leftarrow A[v_L, g_L]$.
(4.2) Otherwise, if the entry $A[v_L, g_L]$ is empty, the procedure is invoked recursively on $(v_L, \bar{s}_L)$. The returned cost x_L is stored in $A[v_L, g_L]$.
(4.3) If the entry $A[v_R, g_R]$ is not empty, then $x_R \leftarrow A[v_R, g_R]$.
(4.4) Otherwise, if the entry $A[v_R, g_R]$ is empty, the procedure is invoked recursively on $(v_R, \bar{s}_R)$. The returned cost x_R is stored in $A[v_R, g_R]$.

(5) The procedure returns the lowest value for $\max\{x_L, x_R\}$ among all pairs of functions examined.

THEOREM 3.7. *For a binary tree with n nodes, the preceding algorithm returns the cost of the optimal hotlink assignment in polynomial time.*

PROOF. The correctness follows from the correctness of Proc^w and from Lemma 3.6.

For the time complexity analysis, we note that the dynamic programming table has size $O(n2^{4\log n+2}) = O(n^5)$, since for every node v, the maximum number of v-separated functions with distinct signatures is $2^{4\log n+2}$ and we have at most n possibilities for v. Furthermore, each invocation requires us to go over all possible partitions of the set $\mathcal{P}_v^I$ that do not generate crossing links. If a node $u \in \mathcal{P}_v^I$ does not belong to $D_S(v)$, then we do not need to associate u with one of the sets B_L, H_L, B_R, or H_R in step 2 of the aforementioned algorithm because it would violate the well-formedness property. Thus, for a function S, we need only go over all possible partitions of the set $\mathcal{P}_v^I \cap D_S(v)$, whose cardinality is at most $4\log n + 2$. Since the number of these partitions is $O(2^{4\log n+2}\log^2 n)$ and $O(n)$ is required to generate the corresponding hotlink functions for each of them, it follows that the algorithm runs in $O(n^{10}\log^2 n)$. □

Let us note that an optimal set of hotlinks can be obtained by following the same approach employed by procedure Find-Solution described in Section 2.3.

In addition, since the techniques employed in Sections 2.4 and 2.5 can be adapted to work here, we can state the following theorem.

THEOREM 3.8. *There exists a polynomial-time algorithm for solving the hotlink enhancement problem with unknown probability distribution on n-node trees where every node can have up to $K = O(1)$ hotlinks directed from it.*

Finally, we remark that the aim of this section was only to show that the version of the hotlink enhancement problem with unknown probabilities can be solved in polynomial time. In fact, the running time can be improved through a more careful analysis that shows that the constant 4 of the logarithmic bound derived in Lemma 3.4 can be reduced to 2.89. For additional details, we refer to Pessoa et al. [2004].

4. EXPERIMENTS

In this section, we present experiments performed with the hotlink enhancement problem on actual websites. The motivation is twofold: first, to show that adding hotlinks to websites indeed provides considerable gains in terms of the user-expected path length; second, to argue that the algorithm proposed in this

Table I. Summary of Trees

	Min	Max	Mean	Standard Deviation
n	32	512484	9488	55980
m	23	493048	8726	53814
Δ	10	3154	203	397
H	3	16	5.9	2.4
$f(T, p, \emptyset)$	1,91	11,99	4,25	1,8

Here, the meanings for the first column are number of nodes (n), number of leaves (m), maximum outdegree (Δ), height (H), and expected path length ($f(T, p, \emptyset)$).

work for the hotlink enhancement problem (with known probabilities) allows a practical implementation.

Our experiments were executed with an optimized implementation of the algorithm described in Section 2.3, denoted here by PATH. For the sake of comparison, we also run experiments with other two known heuristics (*greedyBFS* and AHA) for the hotlink enhancement problem [Czyzowicz et al. 2003; Kranakis et al. 2004]. It shall be mentioned that since no other efficient exact algorithm for this problem was known before, we were able to verify, for the first time, the actual quality of the solutions provided by these heuristics, that is, how their objective values differ from the optimal.

In Section 4.1, we describe the employed instances. Heuristics *greedyBFS* and AHA are described in Section 4.2, briefly. Finally, we present the experimental results in Section 4.3.

4.1 Instances

In order to obtain our instances, we adopted the same approach employed by Czyzowicz et al. [2003]. First, we obtained the directed graphs associated with 85 Brazilian and American university websites. Let $G^1, \ldots, G^{85}$ be these graphs and let s_i, for $i = 1, \ldots, 85$, be the node that models the homepage of the ith site. Next, for $i = 1, \ldots, 85$, we executed a breadth-first search in G^i, starting at s_i, to extract a tree T^i. The motivation to employ trees generated by breadth-first search, among many other possible trees, is the greedy user model; greedy users are expected to traverse the nodes of a DAG according to a breadth-first search. This process generated 85 trees. The main parameters of these trees are summarized in Table I.

As we have access to the webserver logs of the domain *puc-rio.br*, we could perform experiments using real information about the popularity of the pages in this domain. We used one week, access log statistics to build the instance.

To the other 84 sites we applied Zipf's distribution, where the probability of the ith most probable leaf is given by $p_i = \frac{1}{iH_m}$. Here, H_m is the mth harmonic number $H_m = \sum_{j=1}^{m} \frac{1}{j}$. Using Zipf's distribution is motivated by the work of Glassman [1994], which experimentally proves that the popularity of webpages can actually be modeled with Zipf's popularity law. We note that the popularity order among leaves was selected at random.

4.2 Approximation Algorithms

The *greedyBFS* algorithm was designed by Czyzowicz et al. [2003]. The authors defined the *gain* of a hotlink (u, v) as $g(u, v) = p(T_v)(d(v) - d(u) - 1)$, where $p(T_v)$ is the sum of the probabilities of the leaves in T_v and $d(v)$ is the depth of v. First, GreedyBFS assigns a hotlink from the root r of the input tree to the node u that maximizes the gain $g(r, u)$. This assignment generates a DAG which can be converted into a tree, say $T^{(r,u)}$, by removing its edge (u', u), where u' is the parent of u in T (recall Figure 3). Then, the algorithm recursively assigns hotlinks to the subtrees rooted at the children of r in $T^{(r,u)}$.

The *ApproximateHotlinkAssignment* algorithm (AHA for short) was proposed by Kranakis et al. [2004]. The only difference between this algorithm and the previous one is the criterion used to choose the node u. In order to define u, AHA starts at the root of r and walks down over the tree always selecting, among the children of the current selected node, that child which is the root of a tree with the largest probability. It stops when it finds a node u such that $\frac{p(T_r)}{\Delta+1} \leq p(T_u) \leq \frac{\Delta . p(T_r)}{\Delta+1}$. If such a node either does not exist or is a child of r, then AHA chooses the grandchild of r with the greatest probability.

4.3 Experimental Results

All of our experiments were executed in a Xeon 2.4 GHz machine with 1GB of RAM memory. In addition, we allowed at most 1 hotlink leaving each node in all experiments.

In order to have a clear idea of the improvement produced by an algorithm, we present our results in terms of the gain metric. The gain of an algorithm *Alg* for an instance (T, p) is given by $(f(T, p, \emptyset) - f(T, p, S))/f(T, p, \emptyset)$, where S is the set of hotlinks that *Alg* adds to the tree T.

4.3.1 *Hotlink Assignment to the* puc-rio.br *Domain.* Of special interest in the instance obtained from PUC's domain is the fact that it has actual access probabilities. We note that this instance is one of the biggest that we tested, with 17,379 nodes and $H = 8$.

The optimal solution obtained by PATH provides a gain of 18.62%. PATH executed in less than 2 seconds and consumed around 10MB of memory space. The greedyBFS strategy also performed very well, since it provided a solution with 18.37% gain in less than 0.1 second. On the other hand, the solution provided by AHA, although quickly generated, provided a gain of only 4.95%.

4.3.2 *Hotlink Assignment to Actual Websites with Zipf's Distribution.* A critical aspect of PATH is the use of memory for the dynamic programming table. With 23MB, PATH provided the optimal solution of 82 out of 84 instances. The 2 hard instances (not solved with 23MB) were those with the greatest heights. One of the hard instances has 10,484 nodes and height 16, requiring 488MB. The other instance has 512,484 nodes and height 14, requiring more than 1GB.

Table II summarizes the gains and execution times of greedyBFS, AHA, and PATH for the 82 instances that were solved by PATH using at most 23MB of memory. For PATH, Table II also shows the memory consumption. We note that for these instances PATH has good performance in terms of execution time and memory consumption.

Table II. Summary of Results for the 82 Instances Solved Using up to 23MB

	Min	Max	Mean	Std. Dev.
Optimal Gain (PATH)	7.6	51.9	27.75	10.97
Gain - *greedyBFS*	7.6	51.9	27.36	10.82
Gain - AHA	0.1	31.5	9.74	8.55
Time - *greedyBFS*	0.01	0.15	0.016	0.019
Time - AHA	0.01	0.14	0.015	0.017
Time - PATH	0.01	6.35	0.286	0.980
Memory (Kb) - PATH	3	23063	1493	3960

Time in seconds and memory in KB.

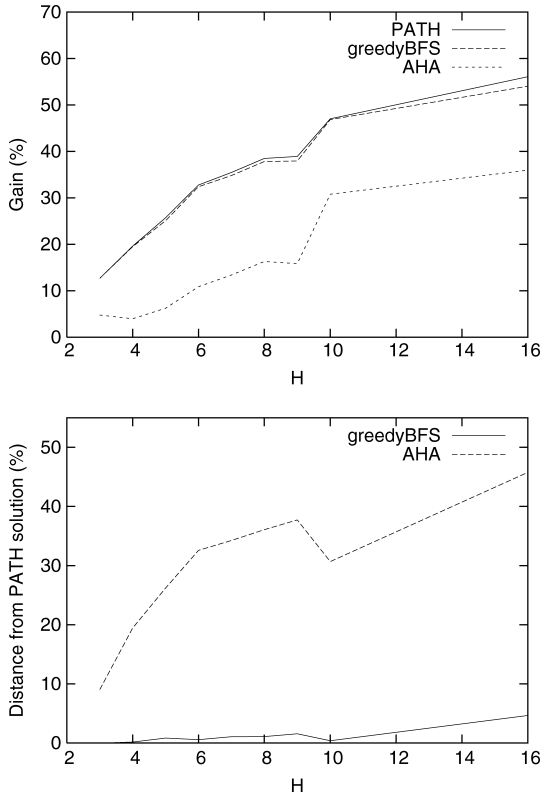

Fig. 4. Average gain (%) due to each algorithm as a function of height H (left), and relative difference between the solution cost of each algorithm and that of PATH (right).

The first chart in Figure 4 shows the average gain of each algorithm as a function of input height. The second chart shows how the solution cost of each algorithm compares to that of PATH. We observe that the solutions of both greedyBFS and AHA become farther from the optimal ones as H increases. We also observe that, as in Czyzowicz et al. [2003], the solutions given by greedyBFS are better than those of AHA. Finally, we note that the greedyBFS solutions are at most 5% from the optimal ones (on average). Learning the final

aforementioned point was only made possible by the fact that PATH computes the optimal hotlink assignment.

5. CONCLUSION

We have presented new exact algorithms for several variations of the problem of assigning hotlinks to hierarchically arranged data such as in web directories. For most of these variations, we have proved that the proposed algorithms run efficiently from a theoretical point of view. In the case of one of the algorithms, the running time is polynomial if the depth of the tree is logarithmic. In cases we observed of real-life applications, there are usually several items (i.e., children) in each category (i.e., parent). Hence, the assumption of a logarithmic depth seems reasonable.

We have run some experiments to evaluate both the efficiency and efficacy of the algorithm that solves the problem for known distributions and at most one hotlink leaving each node. These experiments show that significant improvement in the expected number of accesses per search can be achieved in websites using this algorithm. In addition, the proposed algorithm consumed a reasonable amount of computational resources to obtain optimum hotlink assignments. Last, but not least, let us mention the following contribution of the article: Our experiments indicate that simple and very fast greedy heuristic proposed in Czyzowicz et al. [2003] performs very well. Thus, if a worst-case guarantee is not a major issue, then this simple heuristic turns out to be a good alternative.

REFERENCES

ARMSTRONG, R., FREITAG, D., JOACHIMS, T., AND MITCHELL, T. 1995. WebWatcher: A learning apprentice for the World Wide Web. In *Working Notes of the AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*. Stanford, CA, AAAI Press, 6–12.

ATTARDI, DI MARCO, S., AND SALVI, D. 1998. Categorization by context. *J. Universal Comput. Sci. 4*, 9, 719–736.

BOSE, P., KRIZANC, D., LANGERMAN, S., AND MORIN, P. 2002. Asymmetric communication protocols via hotlink assignments. In *Proceedings of the 9th Colloquium on Structural Information and Communication Complexity*. 33–39.

BOSE, P., CZYZOWICZ, J., GASIENIEC, L., KRANAKIS, E., KRIZANC, D., PELC, A., AND MARTIN, M. V. 2000. Strategies for hotlink assignments. In *Proceedings of the 11th International Symposium on Algorithms and Computation* (ISAAC). 23–34.

CYZYOWICZ, J., KRANAKIS, E., KRIZANC, D., PELC, A., AND VARGAS MARTIN, M. 2003. Enhancing hyperlink structure for improving web performance. *J. Web Eng. 1*, 2, 93–127.

DMOZ. 2007. DMOZ website. www.dmoz.org.

FINK, J., KOBASA, A., AND NILL, A. 1996. User-oriented adaptivity and adaptability in the AVANTI project. In *Designing for the Web: Empirical Studies*. Microsoft Usability Group, Redmond, WA.

GERSTEL, O., KUTTEN, S., MATICHIN, R., AND PELEG, D. 2003. Hotlink enhancement algorithms for web directories. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*. 68–77.

GLASSMAN, S. 1994. A caching relay for the World Wide Web. In *Proceedings of the 1st International World Wide Web Conference*. 69–76.

GOOGLE. 2007. Google website. http://www.google.com/.

KRANAKIS, E., KRIZANC, D., AND SHENDE, S. 2004. Approximating hotlink assignment. *Inf. Proc. Lett. 90*, 3, 121–128.

MATICHIN, R. AND PELEG, D. 2003. Approximation algorithm for hotlink assignments in web directories. In *Proceedings of the 8th Workshop on Algorithms and Data Structures*. Ottawa, Canada, 271–280.

PERKOWITZ, M. AND ETZIONI, O. 1999. Towards adaptive web sites: Conceptual framework and case study. In *Proceedings of the 8th World Wide Web Conference*.

PESSOA, A., LABER, E., AND SOUZA, C. 2004a. Efficient implementation of a hotlink assignment algorithm for web sites. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*.

PESSOA, A., LABER, E., AND SOUZA, C. 2004b. Efficient algorithms for the hotlink assignment problem: The worst case search. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC)*.

YAHOO. 2007. Yahoo website. http://www.yahoo.com/.

Received May 2005; revised February 2007; accepted March 2007