An Experimental Study of Recent Hotlink Assignment Algorithms

Tobias Jacobs*

Abstract

The concept of *hotlink assignment* aims at enhancing the structure of web sites such that the user's expected navigation effort is minimized. We concentrate on sites that are representable by trees and assume that each leaf carries a weight representing its popularity.

The problem of optimally adding at most one additional outgoing edge ("hotlink") to each inner node has been widely studied. A considerable number of approximation algorithms have been proposed and worst-case bounds for the quality of the computed solutions have been given. However, only little is known about the practical behaviour of most of these algorithms yet.

This paper contributes to close this gap by evaluating all recent strategies experimentally. Our experiments are based on trees extracted from real websites as well as on synthetic instances. The latter are generated by a new method that simulates the growth of a web site over time. We also propose a memory-efficient way to implement an optimal hotlink assignment algorithm, making it possible to compute optimal solutions for larger instances than before. Finally, we present a new approximation algorithm that is easy to implement and exhibits an excellent behaviour in practice.

1 Introduction

The design of web sites typically aims at making a large amount of information conveniently accessible. Web designers cannot arbitrarily distribute the contents among the pages as this would make information retrieval too complex for the users. The site's structure must rather somehow represent structural properties of the information. On the other hand, the interests of users on most web sites are highly correlated. Typically, about 80% of the users access only about 20% of the pages (cf. [14]). Moreover, the popularity of pages is likely to change over time.

By automatically moving popular pages closer to the home page one can both reduce web traffic and increase user-friendliness. However, restructuring the whole web site is not practicable for reasons mentioned above. In contrast, the concept of adding additional hyperlinks called *hotlinks* to the pages is a non-destructive approach which preserves the original site's structure.

We concentrate on web sites that are representable by a rooted tree, where the root is the home page, inner nodes represent navigation pages and the information is contained in the leaves, each having a certain popularity. This model has been widely studied in literature ([2, 3, 4, 5, 6, 9, 10, 11, 12, 13]) and a number of algorithms with provable good worst-case behaviour have been proposed. Our main purpose is to evaluate these algorithms experimentally and to make a statement about which of them are recommendable in practice.

Problem definition. A weighted tree T is a triple (V, E, ω), where (V, E) is a tree rooted at a node $r \in V$. Let $L \subseteq V$ be the set of leaves. The weight function $\omega : L \to \mathbb{R}_0^+$ assigns a non-negative weight to each leaf.

For nodes $u \in V$ we also write $u \in T$. The set of ancestors (descendants) of u, not including u itself, is denoted $\mathrm{anc}(u)$ ($\mathrm{desc}(u)$), $\mathrm{ch}(u)$ is the set of u's direct children and $\mathrm{par}(u)$ denotes the parent of u.

A *hotlink* (u, v) is an additional directed edge between nodes in T. We say that u is the *hotparent* of v and v is the *hotchild* of u. We further say that the hotlink *starts* in u and *ends* in v. A set $A \subset V \times V$ of hotlinks is called a *hotlink assignment (HLA)*.

We assume that a user only knows about hotlinks that start in nodes she has already visited, and she always uses any hotlink taking her closer to her destination leaf. This natural behaviour is called the *greedy user model* or *obvious navigation assumption*.

Referring to [13], a hotlink assignment A is called *feasible* if it satisfies the following properties:

(i) $v \in \mathrm{desc}(u)$ for any $(u, v) \in A$.

(ii) If $(u, v) \in A$, then there is no $(u', v') \in A$ with $u' \in \mathrm{desc}(u) \cap \mathrm{anc}(v)$ and $v' \in \mathrm{desc}(v)$.

(iii) For any node $u \in T$ there is at most one $(u, v) \in A$.

Properties (i) and (ii) exclude hotlinks that would never be taken by a greedy user. Property (iii) reflects the requirement that the number of hotlinks on a concise web page must be somehow limited. However, many algorithms naturally generalize to relaxations of that property. In the remainder of the paper, when talking about hotlink assignments, we always mean feasible HLAs.

*Department of Computer Science, University of Freiburg.

A hotlink assignment A is optimal if the *path length*

$$p(A) = \sum_{l \in L} \omega(l) \text{dist}^A(r, l)$$

is minimal among all hotlink assignments for the tree under consideration, where $\text{dist}^A(r, l)$ denotes the length of the greedy user's path from r to l in the graph $(V, E \cup A)$. We abbreviate $\text{dist}^{\emptyset}(u, v)$ by $\text{dist}(u, v)$. The task of finding an optimal hotlink assignment for a given tree is called the *Hotlink Assignment Problem.*

Equivalently, one can also maximize the *gain*

$$g(A) = p(\emptyset) - p(A) \ .$$

In spite of equivalence with respect to optimal solutions, the two optimization terms are not equivalent when considering approximation ratios. Most known polynomial algorithms for the hotlink assignment problem approximate only one of these terms up to a good ratio (cf.Table 3). On the other hand it is possible to combine the approximation properties of several algorithms by subsequently running each of them and then choosing the best among the computed hotlink assignments.

Related work. The hotlink assignment problem has first been formulated by Bose et. al. in [2]. They have shown that is is NP-hard when considering general graphs instead of trees.

To the author's best knowledge it is yet unknown if the optimal solution for trees can be computed in polynomial time. An optimal algorithm whose runtime and space requirements are exponential in the depth of the tree (and thus polynomial for trees of logarithmic depth) has been independently discovered by Gerstel et. al. ([7]) and Pessoa et. al. ([12]). An implementation of this algorithm has been confirmed in [13] to be able to find optimal solutions for trees having ten thousands of nodes and a depth up to 14.

The first approximation algorithm for the hotlink assignment problem has been presented in [10]. As in [4] and [13] that algorithm has already been shown to be outperformed by greedy-like strategies, we do not include it in our experiments.

Algorithm GREEDY has first been formulated in [3]. In [9] we have proven that it achieves at least half of the gain of an optimal solution in the greedy user model. Another 2-approximation in terms of the gain has been presented by Matichin and Peleg in [11]. Their algorithm assigns only hotlinks that bypass exactly one node. In [9] we have generalized this approach by showing that restricting hotlinks to some length h leads to a guaranteed approximation ratio of $\frac{h}{h-1}$ in terms of the gain. Using a modified version (called LPATH) of the optimal algorithm mentioned above, such hotlink assignments can be computed in polynomial time for constant h, so LPATH is a PTAS. The abovementioned algorithm of Matichin and Peleg is implicitly covered by the evaluation of LPATH for different values of h, including $h = 2$. By assigning the tree's depth to h we also obtain an efficient method to compute optimal solutions.

Another family of recent algorithms is more tailored to approximate the resulting path length. A lower bound for that optimization term has been given in [2]: Let Δ be the outdegree of the tree under consideration. Then no hotlink assignment can achieve a path length better than $H(\omega)/\log(\Delta+1)$, where $H(\omega) = \sum_{l \in L} \omega(l) \log(1/\omega(l))$ is the entropy of the probability distribution among the leaves (assuming that the weights are normalized to sum up to 1). Douïeb and Langerman have presented two algorithms that guarantee a path length of $O(H(\omega))$ and are thus constant factor approximations for trees of constant degree. In [9] we have introduced a 2-approximation algorithm in terms of the path length for arbitrary trees.

Our contribution. The main intention of this work is to close the gap between theory and practice concerning the hotlink assignment problem. We have implemented all algorithms mentioned above and evaluated their performance by applying them to two different sets of trees. The first set contains instances that have already been used for the tests in [13]. We have extended this set by extracting 13 additional trees from German university web sites. These new instances contain up to 217.000 nodes and are thus about a factor of 5 larger than the instances from the original set. The trees contained in the second set have been generated by a new random construction method that is based on a model of Barabási and Albert ([1]) and generates more realistic instances than the methods used in previous experiments on hotlink assignment ([4]).

We have also implemented and evaluated a completely new approximation algorithm which is based on a heuristic given in [9]. It turns out that this method, in spite of its simplicity, performs excellently in practice. We will see that it is only up to 5% away from the optimal solution for almost all instances, which is only beaten by the PTAS.

Concerning the approximation scheme LPATH, we give an implementation whose memory requirements only depend on the depth of the tree. Thus we have been able to compute optimal solutions for all tree instances considered in [13] within 500MB of RAM. This improves upon the implementation of an optimal algorithm proposed in that work.

This paper is organized as follows: Some further notation is given Section 2. That section also introduces the hotlink assignment algorithms we have evaluated experimentally. Section 3 describes the acquisition of our test instances and the experimental setup. The results of our study are presented in Section 4. Section 5 concludes.

2 Further notation and algorithms

In this section we introduce some further notation for our optimization problem. We then describe the hotlink assignment algorithms we have evaluated in our experiments and discuss some implementation issues. We use the terms *algorithm*, *strategy* and *method* interchangeably throughout the paper.

2.1 Notation For $u \in T$ we denote by $T(u)$ the maximal subtree of T rooted at u. For any set V' of nodes, $T(-V')$ is the maximal subtree of T that is obtained from T by removing all subtrees rooted at an element of V'. Furthermore, for any hotlink assignment A, let $T(u)(-A) = T(u)(-\{v \mid \exists(u', v) \in A, u' \notin T(u)\})$ be the subtree obtained from $T(u)$ by omitting all subtrees rooted at hotchildren of u's ancestors.

We define the weight of an inner node u as the sum over the weights of those leaves having u on the (greedy user's) path from r to them. Note that weights of inner nodes possibly change when hotlinks are assigned or modified. Formally,

$$\omega^A(u) = \begin{cases} \omega(u) & \text{if } u \text{ is a leaf} \\ \sum_{v \in \mathrm{ch}^A(u)} \omega^A(v) & \text{otherwise} \end{cases},$$

where $\mathrm{ch}^A(u)$ is the set of u's children and hotchildren in $T(u)(-A)$. We abbreviate $\omega^\emptyset$ by ω.

Next we introduce an order among the children of a node. The relation "$\succ$" is defined as a total order among siblings such that $\omega(u) > \omega(u') \Rightarrow u \succ u'$ for any pair u, u' of siblings. Ties are broken arbitrarily. The direct successor sibling of u with respect to that order is denoted by $\mathrm{succ}(u)$. The *first child* of an inner node is its unique child having no predecessor.

Finally, we define the *heavy path* of a node u. If u is a leaf, then its heavy path only consists of u. Otherwise, it is the path obtained by appending the heavy path of u's first child to u.

2.2 Algorithms We begin by a number of simple *top-down methods.* The latter term has been introduced in [6] (as well as the term "heavy path") and denotes hotlink assignment algorithms that assign a hotlink (r, v) and then recursively apply themselves to $T(v)$ and all $T(u)(-\{v\}), u \in \mathrm{ch}(r)$. Thus any top-down method is fully characterized by the choice of v.

GREEDY: Algorithm GREEDY has first been studied in [3] (where it is called "recursive"). It is a top-down method where the root's hotchild v is chosen such that $g(\{(r, v)\})$ is maximized.

GREEDY is a 2-approximation in terms of the gain ([9]). It has exhibited the best performance among the approximation algorithms studied experimentally so far ([3, 4, 13]).

H/PH: The H/PH-strategy is also a top-down method. Let h be the node whose weight is closest to $\omega(r)/2$. If $\omega(h) > \alpha\omega(r)$, then h is chosen as the hotchild of r. Otherwise the parent p_h of h becomes r's hotchild. The threshold α is given as the solution of $(\frac{\alpha}{1-\alpha})^{2(1-\alpha)} = \alpha$, i.e. $\alpha \approx 0.2965$.

The H/PH-algorithm has been proposed in [6], where it is proved to guarantee a path length of at most $1.141H(\omega) + 1$. Thus it is asymptotically a $(1.141 \log(\Delta + 1))$-approximation in terms of the path length, where Δ is the outdegree of the tree (cf. Section 1).

In [6] the authors also show that h is always located on the root's heavy path. Applying that observation makes the algorithm considerably faster.

PMIN: In [9], $p_{\min}(T)$ has been defined as the sum over the weights of all nodes in T except the root and except the heaviest child of each node. In this work we use a slightly modified definition: $p_{\min}$ is the sum over the weights of all nodes in T except the root and except the heaviest child of each node *but the root.* Formally,

$$p_{\min}(T) = \sum_{u \in V \setminus (L \cup \{r\})} (\omega(u) - \max_{v \in \mathrm{ch}(u)} \omega(v)) + \sum_{l \in L} \omega(l) .$$

PMIN is a top-down method which chooses the hotchild v of r such that

$$p_{\min}(T(v)) + \sum_{u \in \mathrm{ch}(r)} p_{\min}(T(u)(-\{v\}))$$

is minimized. It is a new algorithm and no bounds for its approximation ratios are known yet.

HEAVY PATH: Algorithm HEAVY PATH has been proposed in [5]. It works as follows: First split the tree into the set of heavy paths. This can be done in linear time by recursively computing the set of heavy paths of the subtrees rooted at the children of r, uniting these sets and appending r to the path containing the first child of r. Then interpret each of these paths as a list of weighted elements. The weight $W(u)$ of each such element u is $\omega(u) - \omega(v_f)$, where v_f is the first child of u in the tree. Then a HLA for each such list $u_1 \ldots u_n$ is computed as follows: Assign a hotlink (u_1, u_i) such that $\sum_{1 \leq j < i} W(u_j)$ and $\sum_{i < j \leq n} W(u_j)$ are both at most

$\frac{1}{2}\sum_{1\leq j\leq n} W(u_j)$ and recursively apply the algorithm to the sublists $u_2 \dots u_{i-1}$ and $u_i \dots u_n$.

The whole algorithm can be implemented such that it takes linear time. Thus the heavy path strategy has the lowest worst-case runtime among all strategies evaluated in this work. It guarantees a maximum path length of $3H(\omega)$, making it a $(3\log(\Delta + 1))$-approximation in term of the path length.

LPATH: The *length* of a hotlink (u, v) is defined as $\text{dist}(u, v)$. The best assignment of hotlinks that have a maximum length of h is a $\frac{h}{h-1}$-approximation in terms of the gain (cf. [9]). Algorithm LPATH, which for any given h computes a HLA that is at least as good as the best length h assignment in worst-case time $O(|V|3^h)$, is thus a PTAS in terms of the gain. In our experiments we also use LPATH as an optimal algorithm by setting h to the tree's depth.

LPATH is a dynamic programming algorithm. Subproblems are determined by a subtree together with a number of hotlinks starting outside that subtree that have already been decided to end in it. More formally, subproblems are defined by a triple (q, a, u), where $q = q_1 \dots q_n$ is a directed path, $a = a_1 \dots a_n a_{n+1} b \in \{0,1\}^{n+2}$ is a binary vector and u is node in T. It represents the tree qT_u obtained by appending the tree T_u to q_n. $T_u = T(\text{par}(u))(-\{u' \mid u' \succ u\})$ (not to be confused with $T(u)$) is the tree obtained by deleting all subtrees from $T(\text{par}(u))$ that are rooted at a sibling $u' \succ u$. The vector a represents a number of restrictions concerning HLAs for qT_u. Path nodes q_i are only allowed to have a hotchild if $a_i = 1$. They are not allowed to be hotchildren themselves. The node $r = \text{par}(u)$ is only allowed to be a hotparent if $a_{n+1} = 1$, and may only be a hotchild if $b = 1$.

For a given subproblem (q, a, u), LPATH deletes the first $|q| - h$ components of q and a in case of $|q| > h$. This only excludes hotlinks that would have a length greater than h. Then all possible configurations of hotlinks starting in nodes from q are compared. A configuration is determined either by one hotlink from q pointing at r, or by a selection of hotlinks starting at nodes in q that end somewhere in $T(u)$. In the latter case the remaining hotlinks end in $T_{\text{succ}(u)}$.

Each such configuration results in new subproblems, see [9] for details. As the solution of (q, a, u) only depends on a and u, the number of subproblems is limited by $|V|2^{h+2}$. Summing up the number of configurations for all subproblems results in a runtime of $O(|V|3^h)$ (cf. [13]).

Next we discuss a number of practical improvements to LPATH. The dynamic programming approach for the algorithm has been adopted from the PATH algorithm of Pessoa. Laber and Souza [12]. The only difference to LPATH is that, whenever the path q of a subproblem becomes longer than h, PATH gives up. In [13] Pessoa et. al. have proposed two improvements to their algorithm. The first is to increase the number of considered hotlink assignments by always cutting the first component from q and a until $a_1 = 1$. That improvement is already included in the original definition of LPATH, as the latter strategy always cuts the first components q and a when they become too long. The second improvement is a consequence of the observation, that for any subproblem (q, a, u) the total number of hotlinks starting from path nodes q_i is limited by the number of leaves in T_u. Let $\hat{a} = \sum_{1\leq i\leq |a|} a_i$ and let l_u be the number of leaves in T_u. We adopt the second improvement by inverting the $\hat{a} - l_u$ components of a having the highest possible indices from 1 to 0 whenever the $\hat{a} > l_u$.

In our experiments we have observed that the memory requirements of LPATH are by far more critical than the runtime is. For all tree instances the algorithm either terminated in reasonable time (2-3 minutes or less), or the memory requirements exceeded our hardware limit. So the purpose of our main improvement is to reduce space consumption.

The total number of subproblems considered by LPATH is $\Theta(|V|2^h)$. Fortunately, we do not need to store all of them simultaneously. For any fixed node u, let S_u be the set of solutions for all possible values of (a, u). Let u_f be the first child of u. Then $S_{\text{succ}(u)}$ and S_{u_f} suffice to compute any element of S_u. Furthermore, $S_{\text{succ}(u)}$ and S_{u_f} are not required for any further computation, so these sets can be removed from memory after computing S_u.

Let $u_1, \dots, u_n$ be the unique postorder sequence of $V - \{r\}$ where each node appears before its successor with respect to "$\succ$". We successively compute $S_{u_1}, \dots, S_{u_n}$ and delete any set as soon as it is not required anymore. It is easy to observe that with this policy any solution of a subproblem is guaranteed to be available when it is required. Note that the solution of the original problem is contained in S_{u_n}. The following lemma bounds the number of solution sets that are simultaneously stored.

LEMMA 2.1. *Let d be the depth of the tree. At any time during the execution of* LPATH, *for $1 \leq x \leq d$ there is at most one node u with $\text{dist}(r, u) = x$ whose solution set S_u is currently stored and not currently computed.*

Proof. Let u and u' be two nodes that have the same distance to the root and let u' occur after u in the postorder sequence. Let $S_{u'}$ be currently stored and completely computed. If u and u' are siblings, then S_u has been used for the computation of $S_{u''}$, where u'' is either u' or another sibling between u and u', so S_u has already been deleted. If u and u' are not siblings, then the parent v of u occurs in the postorder sequence before u'. In that case S_v has already been computed, which implies that S_u has been used and thus deleted. □

As only one solution set is computed at a time, it follows directly from Lemma 2.1 that at most $d+1$ solution sets are simultaneously stored and thus the memory requirements of our implementation are $O(d2^h)$. The improvement is significant in practice, as the depth is typically small compared to the number of nodes (cf. Table 1).

CENTIPEDE: In a *centipede hotlink assignment* only the first child of each inner node is allowed to be bypassed by hotlinks. The best centipede HLA is a 2-approximation in terms of the path length ([9]).

From the centipede restriction follows that for each non-heaviest inner node u of a tree T the partial hotlink assignment for $T(u)$ is independent from that for T', where T' is obtained from $T(-\{u\})$ by appending a leaf of weight $\omega(u)$ to the the parent of u. So, when computing the best centipede hotlink assignment for T, the trees $T(u)$ and T' can be considered separately.

By applying this observation to each non-heaviest inner node, the algorithm CENTIPEDE splits the tree into a set of *centipede trees*, which are trees whose inner nodes have at most one non-leaf child. Optimal hotlink assignments for centipede trees are then computed in polynomial time by a dynamic programming algorithm, see [9] for a detailed description.

3 Experimental setup

3.1 Real instances Our set of real instances consists of 33 trees. 20 of them represent Brazilian university sites and have been made available by the authors of [13]. Their size (depth) ranges between 48 (4) and 57,877 (16).

The remaining thirteen instances represent websites of German universities, ranging from size (depth) 26,194 (5) to 217,213 (88). We have extracted the trees from the corresponding sites using breadth-first search, which implies that, for any page v in the original structure, a shortest path from the home page to v becomes the unique path from r to v in the resulting tree. In order to guarantee that no page occurs more than once, our algorithm maintains two hash tables: One for the web addresses already visited and one for the contents of the corresponding pages.

3.2 Synthetic instances As we want to make reliable statements about the algorithms' behaviour for different tree sizes, we need a large number of test instances. The available 33 trees extracted from web sites are not sufficient for that purpose. In [13], Pessoa et. al. increase the number of instances by considering each subtree of minimum depth 3 that is rooted at a node in one of their original trees. This approach causes strong dependencies in the data set and is therefore problematic in our opinion.

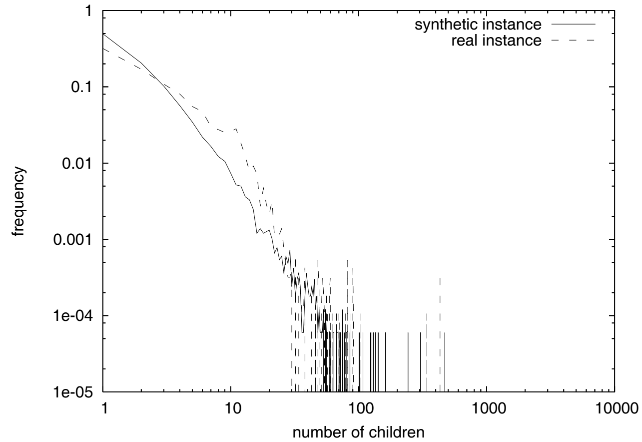

Figure 1: Distribution of the number of children in real and synthetic instance.

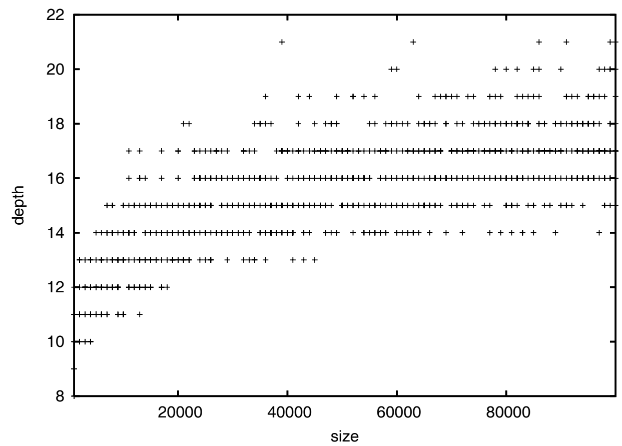

Figure 2: Relation between size and depth in the synthetic trees.

Instead, we randomly generate a large number of synthetic trees. Czyzowicz et. al. ([4]) have observed that the outdegree Δ of inner nodes follows a power-law, i.e. $P[\Delta = i] \sim i^{-\Delta}$. They have chosen the exponent $k = 2.72$ from literature about the graph structure of the World Wide Web. Their proposed algorithm maintains a FIFO-queue where all new nodes are stored. In each iteration step a node is removed from that queue and is assigned a random number i of new nodes as its children, where i is chosen according to the abovementioned power-law. The algorithm terminates as soon as the desired number of nodes has been generated.

In our opinion there are two problems with that method. The first is that the leaves of any possible output tree are basically all on the same level, which is a consequence of the breadth-first and top-down manner of the construction. The second problem is that, according to our given data sets, the chosen value of k does not seem to be adequate. Maximum-likelihood-estimation of that parameter (see e.g. [8]) based on our real instances resulted in $k \approx 1.78$. This value is also problematic as for $k \leq 2$ the expectation of the number of children is infinity, which is unrealistic for trees having a limited number of nodes.

We have therefore developed a new algorithm that is based on a model of Barabási and Albert ([1]). In their model a graph, initially containing a small number of m_0 nodes, is built by in each iteration step i adding a new node v_i adjacent to m existing nodes. The probability for any node v to be chosen as one of v_i's neighbors is proportional to the number of nodes already adjacent to v. The authors show that for large i the number of neighbors of a node converges against a power-law distribution. In our case $m_0 = m = 1$, so that the resulting graph is a tree. Unlike the algorithm of Czyzowicz et. al., our construction generates trees with leaves at all levels.

The data set generated for our experiments consists of trees having sizes $1000, 2000, \ldots, 100000$. For each of these size values ten instances were generated, so their total number is 1000.

In these synthetic trees the distribution of the nodes' outdegree is similar to the typical distribution in real instances, as Figure 1 shows. Figure 2 displays the depth of the generated trees. It turns out that the depth tends to grow only very slowly with the size.

3.3 Assigning weights As no user access patterns are available to us, we have randomly assigned weights to the leaves of both the synthetic and the real trees. We did so using Zipf distribution, i.e. the ith heaviest leaf is assigned a weight of $\frac{1}{iH_m}$, where H_m is the mth harmonic number and m is the number of leaves in the tree. Zipf distribution is considered as the typical access distribution, see e.g. [14]. The same approach has also been employed in [3] and [13].

3.4 Test environment We have run all algorithms described in Section 2 on both the real and the synthetic tree instances. In case of LPATH we have applied each configuration of $h = 2 \ldots 15$.

We have measured the path length of the resulting hotlink assignments as well as the runtime of the algorithms.

$\|V\|$	d	OPT	GREEDY	PMIN	H/PH	HEAVY PATH	CENTI-PEDE
48	6	53.41	53.41	53.41	50.58	49.10	51.14
117	4	21.77	21.77	21.77	14.12	0.00	15.38
320	7	42.73	40.88	42.68	29.68	31.72	41.66
369	5	30.39	30.39	30.25	12.15	12.74	24.13
443	10	49.72	49.54	44.81	37.37	30.97	40.24
542	6	26.53	26.03	24.86	7.15	6.44	22.35
556	4	12.70	10.98	12.70	0.32	0.00	9.69
646	5	28.12	27.26	28.12	14.91	11.99	26.35
746	9	44.74	44.56	44.52	31.24	29.70	41.16
842	10	51.67	50.25	50.70	38.02	33.41	41.58
1016	4	18.35	18.35	18.35	5.76	5.12	17.93
1100	8	46.07	45.96	45.21	27.36	34.14	43.81
1151	7	35.51	35.03	33.63	19.91	19.59	33.35
1158	7	26.53	25.93	25.88	16.81	6.07	22.30
1743	3	13.15	13.15	13.15	6.15	6.12	8.63
1892	9	36.99	35.98	36.28	19.14	22.40	32.98
2275	6	30.37	30.29	30.29	22.18	21.32	27.99
2317	6	21.38	20.75	21.36	8.43	5.57	18.65
10484	16	55.21	54.22	54.41	43.29	41.23	50.24
24986	16	48.22	47.07	47.85	33.27	31.56	45.72
26194	5	23.43	22.77	22.92	11.90	11.64	20.59
30890	56	38.81	37.98	38.14	27.83	25.50	34.54
45439	23	35.89	35.33	35.32	20.95	19.48	32.24
57877	10	39.99	37.20	39.71	25.21	24.74	36.39
78472	45	?	54.80	56.92	47.01	45.95	54.15
89894	41	?	45.86	45.99	32.25	31.02	43.66
96288	26	?	42.61	43.04	29.02	30.47	39.52
107591	37	?	49.12	49.09	39.70	34.29	49.21
110892	58	?	38.87	39.43	26.40	27.04	36.61
115044	42	?	45.27	44.14	28.54	31.23	40.68
120330	88	?	59.23	58.40	45.97	48.45	56.95
163237	40	?	48.28	48.39	34.73	35.39	?
217213	15	43.76	42.32	43.36	31.91	32.03	41.02

Table 1: Relative gain (%) concerning the real instances.

Based on the path length we have calculated a number of additional values. As defined in Section 1, the gain $g(A)$ of an assignment A is $p(\emptyset) - p(A)$. For reasons of comparability between different instances we have also computed the *relative gain* as $g(A)/p(\emptyset)$.

The main focus of our study lies on the approximation ratios that occur in practice. Therefore, for all instances where an optimal solution OPT could be computed by our implementation of LPATH, we have calculated $p(A)/p(OPT)$ as well as $g(OPT)/g(A)$. Observe that for the latter ratio it is irrelevant if the absolute or the relative gain is used.

4 Results

4.1 Relative gain Table 1 gives an overview of the algorithms' results on the real trees. With our improved implementation of LPATH optimal hotlink assignments for most of the trees could be computed in less than 1 hour without exceeding our memory limit of 500MB.

$\lvert V \rvert$	d	LPATH $h=2$	LPATH $h=3$	LPATH $h=4$	LPATH $h=5$	LPATH $h=6$	LPATH $h=7$
48	6	38.07	52.28	53.41	53.41	53.41	53.41
117	4	21.77	21.77	21.77	21.77	21.77	21.77
320	7	31.68	41.25	42.65	42.73	42.73	42.73
369	5	26.11	30.25	30.39	30.39	30.39	30.39
443	10	37.12	45.09	49.29	49.70	49.72	49.72
542	6	23.51	25.95	26.53	26.53	26.53	26.53
556	4	12.70	12.70	12.70	12.70	12.70	12.70
646	5	23.98	28.12	28.12	28.12	28.12	28.12
746	9	34.05	41.57	44.63	44.74	44.74	44.74
842	10	37.80	48.14	51.48	51.67	51.67	51.67
1016	4	18.35	18.35	18.35	18.35	18.35	18.35
1100	8	34.86	44.44	45.62	45.98	46.07	46.07
1151	7	29.20	33.47	35.51	35.51	35.51	35.51
1158	7	24.86	26.53	26.53	26.53	26.53	26.53
1743	3	13.15	13.15	13.15	13.15	13.15	13.15
1892	9	32.28	36.07	36.99	36.99	36.99	36.99
2275	6	28.75	30.37	30.37	30.37	30.37	30.37
2317	6	21.02	21.38	21.38	21.38	21.38	21.38
10484	16	39.49	51.93	55.03	55.19	55.20	55.21
24986	16	35.47	45.69	47.91	48.17	48.21	48.22
26194	5	22.30	23.43	23.43	23.43	23.43	23.43
30890	56	30.89	37.04	38.71	38.80	38.81	38.81
45439	23	28.99	34.10	35.74	35.87	35.88	35.88
57877	10	32.06	38.57	39.84	39.98	39.99	39.99
78472	45	38.61	51.06	55.56	57.29	57.61	57.79
89894	41	34.46	44.40	46.33	46.51	46.55	46.55
96288	26	32.43	40.52	43.18	43.48	43.59	43.59
107591	37	35.26	46.36	49.36	50.39	50.58	50.64
110892	58	30.03	37.12	39.35	39.67	39.71	39.72
115044	42	32.95	41.19	44.52	45.70	45.93	45.93
120330	88	39.23	53.71	57.63	59.16	59.69	59.94
163237	40	34.33	45.51	47.81	48.55	48.63	48.65
217213	15	32.72	41.00	43.41	43.67	43.73	43.75

Table 2: Relative gain (%) of LPATH concerning the real instances.

This significantly improves the best optimal algorithm previously known: EX-PATH, given in [13], even with a greater amount of memory available, failed for some instances that LPATH has been able to handle. However, for some of our new, larger instances, also LPATH could not compute an optimal assignment.

At first sight one can recognize that GREEDY and PMIN by far yield the best results, always being close to the optimum. In contrast, HEAVY PATH, H/PH and CENTIPEDE are far-off from being near-optimal. Interestingly, these are exactly the algorithms tailored to approximate the resulting path length. The performance of LPATH is only comparable to first-mentioned two strategies for values of h greater than 3.

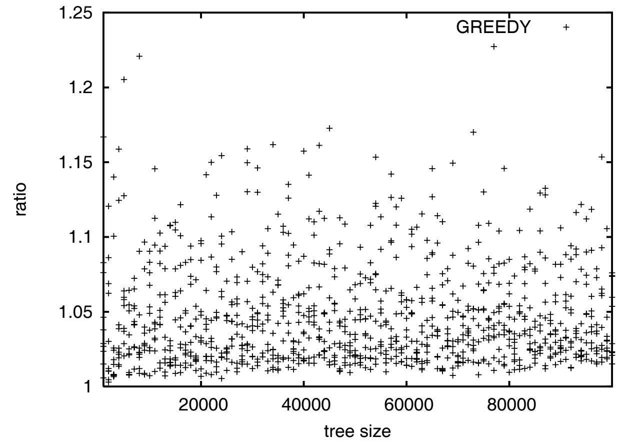

Figure 3: Ratio (gain) of GREEDY and PMIN for different tree sizes.

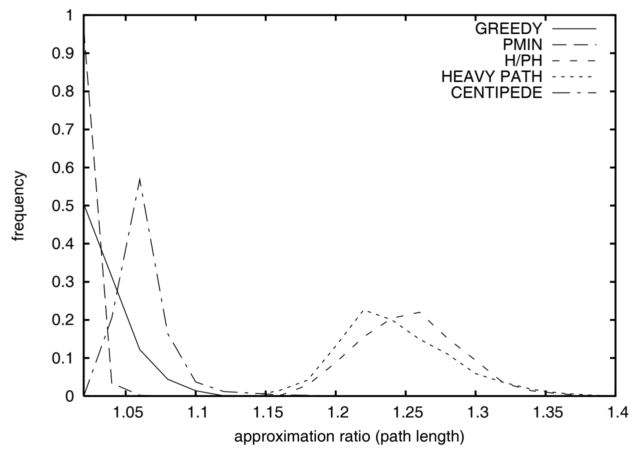

Figure 4: Histogram of the approximation ratio in terms of the path length

4.2 Approximation ratios Table 3 compares the theoretical worst-case approximation ratios to the ratios that have been achieved on the real trees. For algorithms H/PH and HEAVY PATH no bound in terms of the gain had been given yet. However, trees having a depth of 2 where no hotlink at all is assigned by these strategies are easy to construct, so their ratios in terms of the gain do not exist. Almost all algorithms approximate the optimal solution's gain and path length up to the factor 1.5 or better. The only exceptions are H/PH and HEAVY PATH, partially having high factors for the gain.

The experiments based on the random instances allow for more detailed findings. Figure 3 exemplarily shows for the gain of GREEDY that the algorithms' solution quality is basically independent from the size of the tree. The picture looks similar for all algorithms and types of ratios.

The histogram in Figure 5 reveals that, concerning the gain, algorithm PMIN performs even better than GREEDY. For almost all instances PMIN is less and 5%

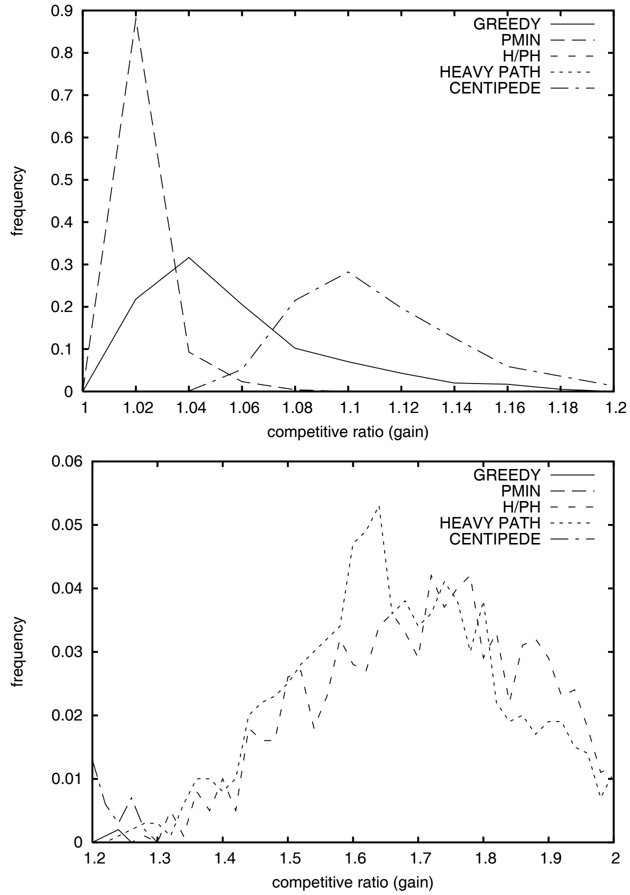

Figure 5: Histograms of the approximation ratio in terms of the gain

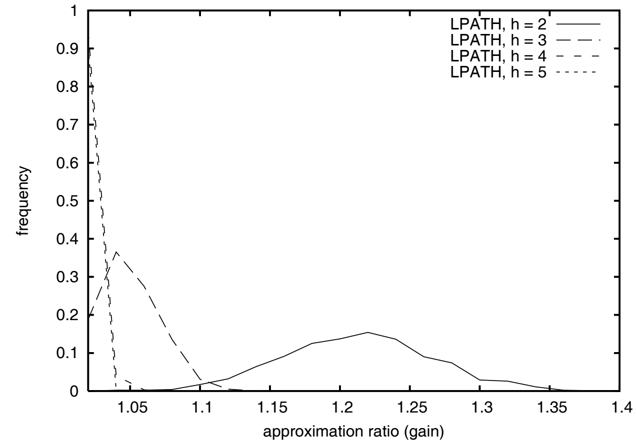

Figure 6: Histograms of the approximation ratio in terms of the gain for LPATH.

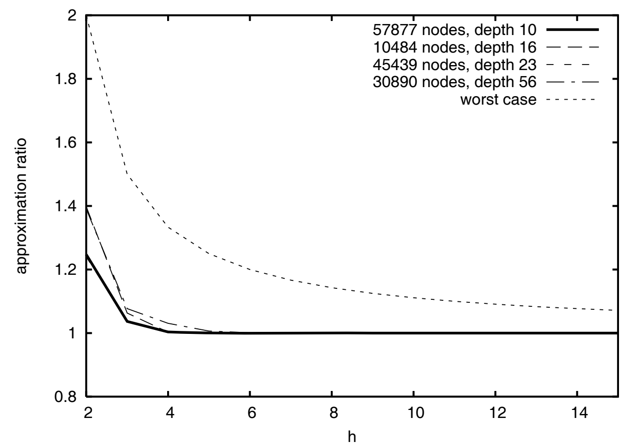

Figure 7: Approximation ratio resulted from running LPATH on tree instances for different values of h .

away from the optimal solution. The plots also show that CENTIPEDE is still quite robust compared to H/PH and HEAVY PATH. All these observations also hold for the path length (Figure 4).

Comparing Figure 5 and 6 confirms that LPATH, for $h = 2$, performs significantly worse than GREEDY. Recall that the former also simulates the approximation algorithm by Matichin and Peleg ([11]) and both strategies are 2-approximations in terms of the gain. For $h = 3$ the results are comparable to those of GREEDY. For $h > 3$ LPATH is very close to the optimal solution. This can also be observed in Figure 7. On each of the four selected instances of real trees the approximation ratio converges to 1 much faster than in theory. However, it seems that the speed of convergence is lower for trees having a greater depth d. Recall that for $h \geq d$ a ratio of 1 is guaranteed.

4.3 Runtime The runtime of LPATH, as expected, grows exponentially with h, up to some characteristic value (Figure 8). For h greater than that value the runtime remains constant. The characteristic value depends on the tree instance. It is typically slightly smaller than the tree's depth, when there are only few possibilities for long hotlinks.

The runtime of the other algorithms for processing the synthetic instances is depicted in Figure 9. The vertical axis is scaled logarithmically here in order to make all runtimes visible in one picture. Apparently, for each algorithm the runtime values form a point cloud that is quite compact, i.e. the runtimes are reliable. For almost all strategies the runtime grows linearly with the tree size, which for most of them is some orders of magnitude better than their worst case. The gradients, however, vastly differ. The fastest algorithm, as expected, is HEAVY PATH, followed by GREEDY, H/PH

	approximation ratio (gain)		
algorithm	worst case	min	max
GREEDY	2	1.000	1.156
PMIN	?	1.000	1.109
H/PH	∞	1.056	39.967
HEAVY PATH	∞	1.088	4.372
CENTIPEDE	∞	1.024	1.524
LPATH, $h = 2$	2	1.000	1.403
LPATH, $h = 3$	1.5	1.000	1.103
LPATH, $h = 4$	1.333	1.000	1.010
	approximation ratio (path length)		
algorithm	worst case	min	max
GREEDY	2	1.000	1.146
PMIN	?	1.000	1.098
H/PH	∞	1.061	1.347
HEAVY PATH	∞	1.081	1.378
CENTIPEDE	∞	1.005	1.209
LPATH, $h = 2$	2	1.000	1.351
LPATH, $h = 3$	1.5	1.000	1.092
LPATH, $h = 4$	1.333	1.000	1.009

Table 3: Theoretical worst-case approximation ratios and experimentally observed ratios. Infinity means that no constant bound exists.

and PMIN. The only algorithm having a superlinearly growing runtime is CENTIPEDE, as the right side of Figure 9 reveals.

5 Summary and conclusion

We summarize the most important findings of our study.

Algorithm PMIN performs excellent on all instances. It is also easy to implement, but its running time is quite high compared to other strategies. The analysis of its theoretical approximation ratios and the development of a more efficient implementation would be an interesting issue for future research.

The GREEDY strategy also exhibits a very good performance. It is slightly worse than PMIN in practice, but the guaranteed approximation ratio of 2 concerning the gain makes this algorithm a good choice. It is also easy to implement and runs faster than PMIN.

Although the path length seems to be the more natural optimization term, strategies tailored to approximate it are not the first choice in practice. The CENTIPEDE algorithm has a very high running time. As the quality of its solutions is only moderate, this algorithm is only advisable in practice if one wants a guaranteed approximation factor of 2 for the path length

Algorithms H/PH and HEAVY PATH exhibit the worst performance among all algorithms studied in this work. Anyhow, HEAVY PATH is extremely fast, making it interesting for quickly computing hotlink assignments for large trees.

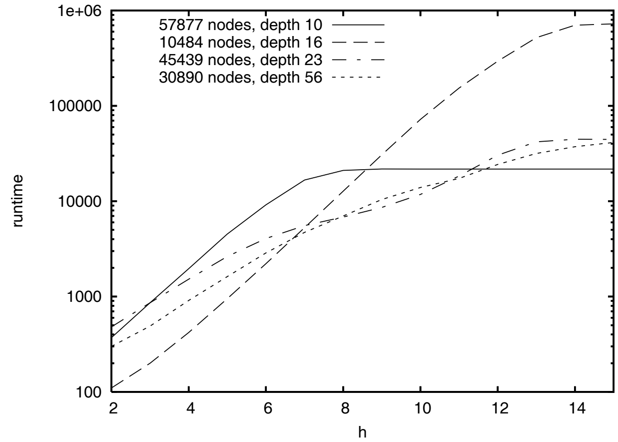

Figure 8: Runtime resulted from running LPATH on tree instances for different values of h .

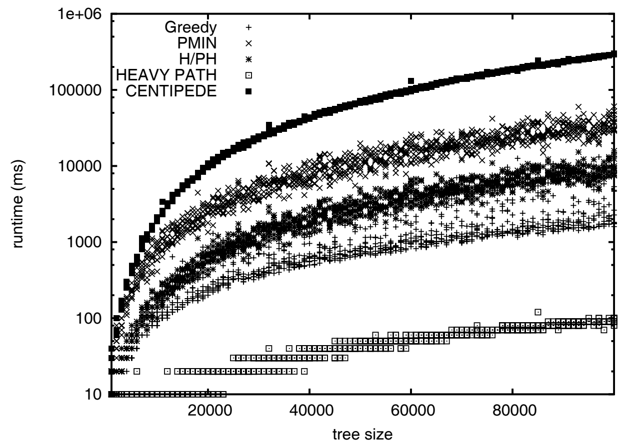

Figure 9: Runtime of the approximation algorithms, logarithmically scaled.

For small values of h the assignments achieved by LPATH are also not as good as the results of GREEDY and PMIN. So this PTAS is only recommendable in practice for $h > 3$.

Note that it is always possible to combine the properties of several algorithms. For example, by running PMIN, GREEDY and CENTIPEDE and then choosing the best among the three resulting assignments one would obtain a solution having the quality of PMIN with a guaranteed approximation ratio of 2 in terms both of the path length and the gain.

Acknowledgments: The author likes to thank Artur Alves Pessoa, Eduardo Sany Laber and Críston de Souza for making the dataset used in [13] available.

References

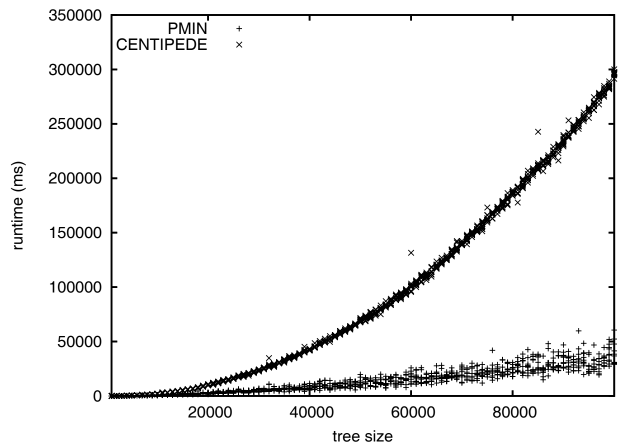

Figure 10: Runtime of CENTIPEDE and GREEDY, linearly scaled.

[1] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[2] P. Bose, E. Kranakis, D. Krizanc, M. V. Martin, J. Czyzowicz, A. Pelc, and L. Gasieniec. Strategies for hotlink assignments. In *International Symposium on Algorithms and Computation*, pages 23–34, 2000.

[3] J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. Martin. Evaluation of hotlink assignment heuristics for improving web access.

[4] J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. V. Martin. Enhancing hyperlink structure for improving web performance. *J. Web Eng.*, 1(2):93–127, 2003.

[5] K. Douïeb and S. Langerman. Dynamic hotlinks. In F. K. H. A. Dehne, A. López-Ortiz, and J.-R. Sack, editors, *WADS*, volume 3608 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 2005.

[6] K. Douïeb and S. Langerman. Near-entropy hotlink assignments. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA 2006)*, volume 4168 of *LNCS*, pages 292–303, Zürich, Switzerland, 2006. Springer Berlin / Heidelberg.

[7] O. O. Gerstel, S. Kutten, R. Matichin, and D. Peleg. Hotlink enhancement algorithms for web directories: (extended abstract). In T. Ibaraki, N. Katoh, and H. Ono, editors, *ISAAC*, volume 2906 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2003.

[8] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. *The European Physical Journal B - Condensed Matter and Complex Systems*, 41(2):255–258, 2004.

[9] T. Jacobs. Constant factor approximations for the hotlink assignment problem. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS 2007)*, Halifax, Canada, 2007.

[10] E. Kranakis, D. Krizanc, and S. Shende. Approximate hotlink assignment. *Inf. Process. Lett.*, 90(3):121–128, 2004.

[11] R. Matichin and D. Peleg. Approximation algorithm for hotlink assignment in the greedy model. *Theor. Comput. Sci.*, 383(1):102–110, 2007.

[12] A. A. Pessoa, E. S. Laber, and C. de Souza. Efficient algorithms for the hotlink assignment problem: The worst case search. In R. Fleischer and G. Trippen, editors, *ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 778–792. Springer, 2004.

[13] A. A. Pessoa, E. S. Laber, and C. de Souza. Efficient implementation of hotlink assignment algorithm for web sites. In L. Arge, G. F. Italiano, and R. Sedgewick, editors, *ALENEX/ANALC*, pages 79–87. SIAM, 2004.

[14] J. E. Pitkow. Summary of www characterizations. *World Wide Web*, 2(1-2):3–13, 1999.