Dynamic Hotlinks

Karim Douïeb* and Stefan Langerman**

Département d'Informatique,
Université Libre de Bruxelles, CP212,
Boulevard du Triomphe, 1050 Bruxelles, Belgium

Abstract. Consider a directed rooted tree $T = (V, E)$ representing a collection V of n web pages connected via a set E of links all reachable from a source home page, represented by the root of T. Each web page i carries a weight w_i representative of the frequency with which it is visited. By adding hotlinks, shortcuts from a node to one of its descendents, we are interested in minimizing the expected number of steps needed to visit pages from the home page. We give the first linear time algorithm for assigning hotlinks so that the number of steps to accede to a page i from the root of the tree reaches the entropy bound, i.e. is at most $O(\log \frac{W}{w_i})$ where $W = \sum_{i \in T} w_i$. The best previously known algorithm for this task runs in time $O(n^2)$. We also give the first efficient data structure for maintaining hotlinks when nodes are added, deleted or their weights modified, in amortized time $O(\log \frac{W}{w_i})$ per update. The data structure can be made adaptative, i.e. reaches the entropy bound in the amortized sense without knowing the weights w_i in advance.

1 Introduction

Since the discovery of the Internet by the general public, the growth of the *World Wide Web* reached an incredible speed and the quantity of information available for all became extraordinary large. By this fact, many inherent problems for consulting of this mass of data appeared, and methods were developed to facilitate and accelerate the search on the web, such as promoting and demoting pages, highlighting links, and clustering related pages in an adaptive fashion depending on user access patterns [15, 6]. In this article we consider the strategy of adding *hotlinks*, i.e. shortcuts from web pages to popular pages accessible from them.

A *web site* can be modeled as a directed graph $\mathcal{G} = (V, E)$ where the nodes V correspond to the web pages and the edges E represent the links. Each node carries a weight representative of its access frequency. We assume that all web pages are reached starting from the *homepage* r. Our goal in adding hotlinks (directed edges from a node to one accessible from it) is to minimize the expected number steps to reach a page from the homepage r.

The idea of hotlinks was suggested by Perkowitz and Etzioni [15] and studied later by Bose et al. [2] who proved that finding the optimal hotlink assignment

* Boursier FRIA, kdouieb@ulb.ac.be
** Chercheur qualifié du FNRS, stefan.langerman@ulb.ac.be

F. Dehne, A. López-Ortiz, and J.-R. Sack (Eds.): WADS 2005, LNCS 3608, pp. 182–194, 2005.

for a DAG is NP-hard, and analyzed several heuristics for assigning hotlinks. More recently, a 2-approximation algorithm for the archivable gain running in a polynomial time was presented by Matichin and Peleg [13].

The problem might become easier when the graph considered is a rooted tree. Kranakis, Krizanc and Shende [12] give a quadratic time algorithm for assigning one hotlink per node so that the expected number of steps to search a node from the root of the tree attain the entropy bound. Several results on adding hotlinks to nodes of d-regular complete trees are also reported by Fuhrmann et al. [8]. Recently, Gerstel et al.[10], and A.A. Pessoa et al. [16] independently discovered a polynomial time dynamic programming algorithm for finding the optimal placement of hotlinks on a tree whose depth is logarithmic in the number of nodes. Experimental results showing the validity of the hotlinks approach are given in [5], and a software tool to structure websites efficiently by automatic assignment of hotlinks has been developed [11].

The concept of hotlinks can be applied to other problems than that of web structuring. For instance, Bose et al.[3] use hotlink assignments to design efficient asymmetric communication protocols. Hotlinks can also be used to design data structures as was demonstrated by Brnnimann, Cazals and Durand [4] with their *jumplist* dynamic dictionary data structure. The jumplist structure can be seen as randomized hotlink assignment on a list, and is meant as a simplification of the skiplist structure [17]. A detreministic version of the randomized jumplist of Brnnimann was developed by Elmasry [7].

In this article, we consider rooted directed trees T with n nodes and maximum degree d. Every node i in T is associated with a weight w_i representative of its access frequency, and $W = \sum_{i \in T} w_i$. Following the *greedy user* model assumption, we assume that the user always takes the hotlink from a node that leads him to a closer point on the path to the desired destination. Due to that, we consider that the assignment of one hotlink which points to a node i can be see as the deletion of any other hyperlink that ends in i. Let T^A be the tree resulting from an assignment A of hotlinks. A measure of the average access time to the nodes is $E[T^A, p] = \sum_{i=1}^{n} d_A(i)p_i$, where $d_A(i)$ is the distance of the node i from the root, and $p =< p_i = w_i/W : i = 1, \ldots, n >$ is the probability distribution on the nodes of the original tree T. We are interested in finding an assignment A which minimizes $E[T^A, p]$.

A lower bound on the average access time $E[T^A, p]$ was given in [2] using information theory [14]. Let $H(p)$ be the entropy of the probability distribution p, defined by $H(p) = \sum_{i=1}^{n} p_i \log(1/p_i)$, then for any assignment of at most δ hotlinks per node the expected number of steps to reach a node from the root of the tree is at least $\frac{H(p)}{\log(d+\delta)}$. This bound is achieved up to a constant factor if $d_A(i) = O(\log(W/w_i))$. We show:

Theorem 1. *Given an arbitrary weighted rooted tree with n nodes and of total weight W. There is an algorithm that runs in $O(n)$ time, which assigns one hotlink per node in such a way that the expected number of steps to reach a node i of weight w_i in the tree from the root is $O(\log \frac{W}{w_i})$.*

This algorithm constitutes a considerable improvement over the previous $O(n^2)$ time algorithm [12]. Furthermore, we present an efficient data structure for dynamically maintaining hotlinks on a tree:

Theorem 2. *There exists a data structure for maintaining hotlinks in a weighted tree T, allowing the insertion and deletion of leaves of weight 1 in T, and the incrementation or decrementation of the weight of any node of T. All updates on a node i of weight w_i run in amortized time $O(\log \frac{W}{w_i})$, and the shortest path to any node i of weight w_i is $O(\log \frac{W}{w_i})$ worst case, where $W = \sum_i w_i$.*

In particular, if the weight of a node is incremented every time that node is accessed, the running time of any sequence of accesses will be bounded by the entropy bound (amortized) without knowing the probability distribution in advance. The proof of the two preceding theorem will be given later in this paper. A weighted and amortized version of the Jumplist data structure is presented in the next section, it is developed for the application of hotlinks assignment to arbitrary trees. In section 3 we give a linear time algorithm to assign hotlinks to trees so that the number of steps to accede to a page from the root of the tree reaches the entropy bound. In Section 4, the dynamic hotlink assignment data structure is described.

2 Jumplists

The data structure named *Jumplist* [4] is a linked list whose nodes are endowed with an additional pointer, the *jump pointer*. Algorithms on the jumplist are based on the *jump-and-walk* strategy: whenever possible use the jump pointer to speed up the search, and walk along the list otherwise. This data structure provides the usual dictionary operations, i.e. SEARCH, INSERT and DELETE.

To each element x of a jumplist is associated a $key[x]$, a $next[x]$ pointer like an ordinary list structure and also an additional $jump[x]$ pointer which points to a successor of x in the list. Note that the jumplists we discuss here only allow insertions/deletions at the end of the list. This restriction greatly simplifies the presentation of the algorithm and is sufficient for its application to the hotlinks problem for trees.

The original version of the jumplists developed by Brnnimann et al. did not consider the access frequencies of the elements of a jumplist. In this paper we develop an enhanced jumplist structure by associating a weight w_x with each element x, proportional to its access frequency, and where the access times reach the entropy bound. In such a situation we would like that the more frequently needed elements be accessed faster than the less frequently needed ones. A measure of the average access time for a jumplist C is

$$\sum_{x \in C} \frac{w_x}{W} d_x,$$

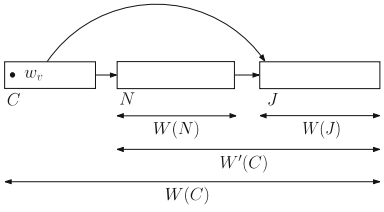

Fig. 1. Weight distribution of a weighted jumplist

where $W = \sum_{x \in C} w_x$ is the sum of the weights of the elements in the sublists of C and d_x is the depth in the jumplist of the element x, i.e. the minimum number of pointers needed to reach element x. We are interested in finding an assignment of the jump pointers for C which minimizes the average access time.

We use the notation $C = (v, N, J)$ for the (sub)list C with first element v followed by the next sublist N, and the jump sublist J. We write $|C|$ for the number of elements in C, and $W(C)$ for the sum of the weights of the elements contained in C, i.e. $W(C) = w_v + W(N) + W(J)$. Also, $W'(C) = W(C) - w_v$. See figure 1.

2.1 Jump Pointer Assignment

Lemma 1. *Given an arbitrary weighted jumplist containing n elements of weights $w_1, \ldots, w_n$, and $W = \sum_{i=1}^{n} w_i$. There is an algorithm which in $O(n)$ time assigns the jump pointers for the jumplist in such a way that $d_i \leq \lfloor \log_2 \frac{W}{w_i} \rfloor + 1$ for $i = 1, \ldots, n$.*

Proof. Constructing a jumplist from a list with weighted elements consists in choosing the jump pointer of the header, and recursively building the next and jump sublists. In order to reach the entropy bound, the jump pointer of the header will point to an element splitting the list into two sublists of roughly equal weight. In other words, the sum of the weights of the elements belonging to the next sublist and that of the jump sublist must be at most equal to half of the total weight of the sublists. Thus for the jumplist $C = (v, N, J)$, the condition of the weighted jumplists will be:

$$W(N) = \sum_{i=2}^{k-1} w_i \leq W/2 \qquad \text{and} \qquad W'(J) = \sum_{i=k+1}^{n} w_i \leq W/2. \tag{1}$$

The problem is to efficiently determine a good element k that satisfies the condition. For this, we first build a table from the jumplist, in time $O(n)$. The i^{th} entry of the table will correspond to the i^{th} element in the jumplist and will contain the value $s_i = \sum_{j=1}^{i} w_j$. Thus, the table is sorted in increasing order and has distinct elements.

Once the table built, we can use exponential search to find the k^{th} element satisfying the condition (1). After this element is found, it will be necessary to

reiterate the process recursively on both sublists. Note that it is not necessary to rebuild tables for that, it is sufficient to use segments of the table built during the first stage.

To improve the speed of the search in the table for the element satisfying the condition (1), we carry out a double exponential search in parallel from both sides of the table, in time $O(\log(\min\{k, n-k+1\}))$, where k is the position of the sought element. We can consequently express the complexity of this algorithm, after construction of the table, by the recurrence $t(n) = O(\log(\min\{i, n-i+1\})) + t(i-1) + t(n-i) = O(n)$. □

2.2 Dynamic Weighted Jumplists

One way to dynamize the weighted jumplist is to use the concept of tolerance. It is a method which consists in requiring that any jumplist $C = (v, N, J)$ satisfies the following relaxed version of condition (1):

$$W(N) \leq W'(C)(1+\tau)/2 \qquad \text{and} \qquad W'(J) \leq W'(C)(1+\tau)/2 \qquad (2)$$

where $0 < \tau < 1$ is a constant tolerance factor. The condition of the tolerant weighted jumplists, eq.(2) described above will be checked recursively by the sublists N and J.

The methods used here are similar to the ones used by Elmasry [7], except for the fact that our structure doesn't need extra informations to maintain the jumplist (such as the number of elements in the next and the jump sublist of each element in the jumplist) and the elements are weighted. The only extra values the dynamic data structure needs to remember are W, the total weight of the entire jumplist, and max_W, the maximal value of W since the last time the jumplist was completely rebuilt.

Searching. The basic search algorithms on jumplist are based on the *jump-and-walk* strategy: Whenever possible use the jump pointer to speed up the search, and walk along the list otherwise (if the jumplist is ordered, it will be trivial to determine if the jump pointer improve or not the speed of the search. Else, in the case where all the elements are arbitrary ordered, we make the assumption that the user knows implicitly which pointer is good to use). For a tolerant weighted jumplist that observes the condition eq.(2), we can determine an upper bound to the number of steps to reach an element from the header of the jumplist.

Theorem 3. *Consider an arbitrary tolerant weighted jumplist C of tolerance factor τ whose total sum of the weights of the elements is W, the number of steps to reach an element i from the header of the jumplist is at most*

$$\lfloor \log(W/w_i)/\log(2/(1+\tau)) \rfloor + 1.$$

Proof. We know that the jumplist C and all its sublists observe the condition of the tolerant weighted jumplist. Let us define $C_k = (v_k, N_k, J_k)$ as the sublist considered after the k^{th} step of the search for element i. That is, $C_0 = C$, and

if at step k the element i is in the next sublist then $C_k = N_{k-1}$, otherwise $C_k = J_{k-1}$. The value of $W'(C_k)$ can be bounded as a function of $W'(C_{k-1})$ using equation (2):

$$W'(N_{k-1}) \leq W(N_{k-1}) \leq W'(C_{k-1})\frac{1+\tau}{2} \text{ and } W'(J_{k-1}) \leq W'(C_{k-1})\frac{1+\tau}{2}$$

so

$$W'(C_k) \leq \max\{W'(J_{k-1}), W'(N_{k-1})\} \leq W'(C_{k-1})(1+\tau)/2.$$

The resolution of the recurrence gives $W'(C_k) \leq W\left(\frac{1+\tau}{2}\right)^k$. Step k of the algorithm will not be performed unless, $w_i \leq W'(C_{k-1}) \leq W\left(\frac{1+\tau}{2}\right)^{k-1}$. This implies that the number of steps k is bounded by $k \leq \log(W/w_i)/\log(2/(1+\tau)) + 1$. □

Thus, the maximum depth an element x can have in a tolerant weighted jumplist of weight W and tolerance factor τ is

$$d_\tau(x, W) = \lfloor \log(W/w_x)/\log(2/(1+\tau)) \rfloor + 1.$$

An element x for which the depth exceeds the value $d_\tau(x, W)$ will be called a *deep* element. The presence of a deep element clearly implies that the jumplist does not satisfy the condition eq.(2).

Insertion We here describe how to insert an element of weight 1 at the end of the list. The insertion operation first uses the jump-and-walk algorithm to find the position of the last element in the jumplist. In the following, we will consider the search sequence $(C = C_0, C_1, \ldots, C_k)$, with $C_j = (x_j, N_j, J_j)$ and $C_{j+1} = J_j$ reaching the last element x_k of the jumplist in k steps (and so N_k and J_k are empty). During an insertion of a new element z, the element is placed in N_k, we increment W, and update max_W to the maximum of W and max_W. If the newly inserted element is deep, i.e. $k + 1 > d_\tau(z, W)$, then one of the lists C_j containing it does not satisfy eq.(2). We reassign the jump pointers of the jumplist as follows. We climb the jumplist, examining x_k, x_{k-1},... until we find an element x_i whose sublist C_i does not satisfy the condition (2). Since x_k is at the end of the jumplist, $W(N_k) = 1$, and $W(C_k) = w_{x_k} + 1$. We compute $W(C_j)$ using the formula $W(C_j) = w_{x_j} + W(N_j) + W(C_{j+1})$, where $W(N_j)$ is computed in time $O(|N_j|)$ by walking the list from element x_j to x_{j+1}. Thus the total cost for finding x_i is $O(|C_i|)$.

We call x_i the *scapegoat* element in reference to the lazy rebalancing schemes for binary search trees developped independently by Andersson and Lai [1], and by Galperin and Rivest [9]. Once the scapegoat element x_i is found, we have to verify that the reconstruction of its sublist C_i will not create deep nodes. If $i \leq \log(W/W(C_i))/\log(2/(1+\tau))$, then the reconstruction of the jump pointers in C_i will not introduce new deep nodes since the number of links to follow from x_i to any element y will be at most $\log(W(C_i)/w_y)/\log(2/(1+\tau))$ in the reconstructed structure. The jump pointers of the sublist C_i can then be reassigned in time $O(|C_i|)$ using Lemma 1. Otherwise, we continue the search for another scapegoat node $x_{i'}$ with $i' < i$.

Let us now consider a sequence of insert operations in a tolerant weighted jumplist whose total weight is W, we wish to show that the amortized complexity per insert is $O(\log(W/w_i))$. We begin by defining a nonnegative *potential function* for the jumplist. Consider the sublist $C = (v, N, J)$, and let $\Phi(v) = max(0, W(N) - W(C)/2, W(J) - W(C)/2)$ be the potential of the element v. We see that an element whose sublists are perfectly balanced have a potential of 0, and an element that does not satisfy the condition eq.(2) have a potential of $\Omega(W(C))$. The potential of the jumplist is the sum of the potentials of its elements.

It is easy to see that by increasing their cost by only a constant factor, the insertion operations pay for the increase in potential of the elements. That is, whenever we pass by an element x to insert a new element as a descendant of x, we can pay for the increased potential in x that may be required by the resulting increase in $\Phi(x)$.

The potential of the scapegoat element x_i, like all the elements that do not observe the condition eq.(2), is $\Omega(W(C_i))$. Therefore, this potential is sufficient to pay for finding the scapegoat element and reassigning the jump pointers of the sublist of which it is the header. These two operations have complexity $O(|C_i|) = O(W(C_i))$.

Deletion. The deletion operation consists of removing the last element of the jumplist, the weight of this element must be equal to 1. We will again use the *jump-and-walk* algorithm to reach this last element. Once the element removed, we update W. Then, if $W < max\_W(1+\tau)/2$, we reassign all the jump pointers of the entire jumplist[C], and we reset max_W to W. If we restate the analysis above ignoring the deletions, the search time is at most $d_\tau(x, max\_W) \leq d_\tau(x, W) + 1$.

Since we perform $\Omega(n)$ operations between two successive rebuilds due to delete operations we can pay for them in the amortized sense (with n equal to the number of elements in the jumplist). Thus for a sequence of delete operations, the amortized complexity per deletion of the last element i of a tolerant weighted jumplist is equal to $O(\log(W/w_i))$.

Reweighting. The reweighting operation allows to increment or decrement the weight of an element by one unit. To find the element to be modified, we again use the jump-and-walk algorithm. Then, for incrementing, we use the same technique as during an insertion: We modify the weight of the element, we check that it does not become deep. If it does, we seek the scapegoat element and we reassign the jump pointers of its sublists. For decrementing, we act as during a deletion: We modify the weight of the element, we update W. Then, if $W < max\_W(1+\tau)/2$, we reassign all the jump pointers of the jumplist, and we reset max_W to W.

The reweight operation is based on the operation of insertion and deletion. We have by this fact same complexities as those, i.e. an amortized complexity per reweight of an element i of a tolerant weighted jumplist equal to $O(\log(W/w_i))$.

3 Hotlinks

The hotlink assignment algorithm for a tree T will proceed by first decomposing the tree into heavy paths (see fig. 2), and then finding hotlink (jump pointers) assignments on the paths viewed as weighted linked lists. In the following, we write T_x for the subtree of T rooted at x, $W(T_x)$ the sum of the weights of all elements in T_x, and $W'(T_x) = W(T_x) - w_x$.

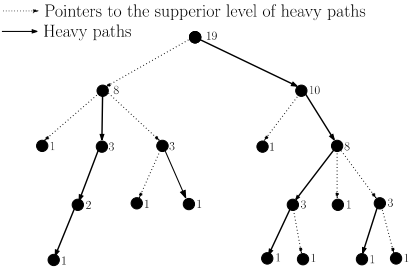

Fig. 2. Example of decomposition of a tree into heavy paths

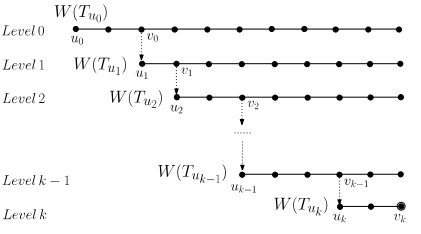

Fig. 3. Example of search into heavy paths of a tree

3.1 Decomposition into Heavy Paths

Following the classical heavy path decomposition scheme [18], we connect each node to its heaviest child, i.e., we pick $next[x]$ among the children of x if

$$W(T_{next[x]}) \geq W(T_y) \ \ \forall y \text{ child of } x. \tag{3}$$

In particular, this implies $W(T_y) \leq W'(T_x)/2 \ \forall y \neq next[x]$ child of x. The chosen edges $(x, next[x])$ naturally decompose the tree into paths. This method of determination of lists is realized in time $O(n)$ with n the number of elements in the tree T, because that decomposition can be done in a single bottom-up tranversal, at the same time as computing the weight of all subtrees.

3.2 Hotlink Assignment

Once the tree is decomposed into heavy paths, we must just apply Lemma 1 to assign the hotlinks (jump pointers) to the paths viewed as jumplists. The weight z_x of an element x in a heavy path will be equal to the weight w_x of the node associated to it plus the sum of the nodes contained in all the subtrees indicated by the children of x except $next[x]$ i.e. $z_x = W(T_x) - W(T_{next[x]})$. The weighted jumplist assignment algorithm is applied on each list using the weights z_x. This method is linear in the number of elements present in each list, so the sum of the assignment complexity for all heavy paths in the tree is $O(n)$. Theorem 4 in the next section shows that this hotlink assignment achieves the entropy bound.

4 Dynamic Hotlinks

In this section we present a data structure for maintaining hotlinks in a weighted tree when leaves are added or deleted and weights modified. Like in the previous section, the tree will be decomposed into paths, and each path will be managed like a jumplist. These jumplists will be managed dynamically using the structure described in section 2. The dynamic determination of the paths in the tree will require some extra work. Indeed, a sequence of update operations can lead the tree to stop satisfying condition (3). Similarily as for the tolerant jumplists, we will use a relaxed version of condition (3):

$$W(T_y) \leq W'(T_x)(1+\tau)/2 \; \forall y \neq next[x] \text{ child of } x, \tag{4}$$

where $0 < \tau < 1$ is the tolerance factor.

Lemma 2. *The tolerant method of decomposition of the tree T into heavy paths guarantee that maximum number of paths visited during a search for node x is at most $\lfloor \log(W/w_x)/\log(2/(1+\tau)) \rfloor$.*

Proof. To determine the maximum number of levels of paths in the decomposition of a tolerant hotlink tree, we must count the maximum number of times k that we can pass from a list to another. Let C_i be the i^{th} list visited during a search. Every time we follow a link from a node x from one list C_i to the head y of another list C_{i+1}, i.e. y is a child of x but $y \neq next[x]$, we know from condition (4) that $W(C_{i+1}) = W(T_y) \leq W'(T_x)(1+\tau)/2 \leq W(C_i)(1+\tau)/2$. The recurrence solves to: $W(C_k) \leq W\left(\frac{1+\tau}{2}\right)^k$ and $w_x \leq W(C_k)$. □

Searching. 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 node. When hotlinks are added, there will exist multiple alternative paths for certain destinations. Again, an underlying assumption at the basis of hotlink idea is that faced with a hotlink in the current node, the user will be able to tell whether or not this hotlink may lead it to a closer point on the path to the desired destination. This has been referred to as the *greedy user* model. Otherwise, we can remark that with the *clairvoyant user* model, we make the assumption that the user somehow knows the topology of the enhanced structure. So, he will always choose the shortest path to reach the desired destination. But, with the method used in this paper, the two models will lead to the same choice of links because no two hotlinks will ever cross. We can now bound the maximum number of steps during a search operation:

Theorem 4. *Consider an arbitrary weighted rooted tree T with W the sum of weights of all its nodes. If one hotlink per node is assigned using a tolerant path decomposition and tolerant jumplists with tolerance $0 < \tau < 1$, then there is a constant a_τ so that the number of steps to reach an element x is at most $d'_\tau(x, W) \leq a_\tau \log(W/w_x)$.*

Proof. The search of an node in a tree, can be seen as a succession of search operations in multiple heavy paths composing the tree. The complexity of a search operation in the heavy paths is given by theorem 3. Consider a search entering the i^{th} path at node u_i, and leaving it at node v_i to enter the $(i+1)^{th}$ path at node u_{i+1}, with u_0 being the root of T and $v_k = x$ is the element we are looking for. See Fig.3. Then number of links followed on the i^{th} path is at most $\lfloor \log(W(T_{u_i})/z_{v_i})/\log(2/(1+\tau))\rfloor + 1$ and $z_{v_i} = W(T_{v_i}) - W(T_{next[v_i]}) \geq W(T_{u_{i+1}})$. So the total number of links followed along heavy paths and hotlinks is at most:

$$\begin{aligned} t &\leq k + [\log \frac{W(T_{u_1})}{z_{v_1}} + \log \frac{W(T_{u_2})}{z_{v_2}} + \cdots + \log \frac{W(T_{u_k})}{z_{v_k}}]/\log(2/(1+\tau)) \\ &\leq k + [\log \frac{W}{W(T_{u_2})} + \log \frac{W(T_{u_2})}{W(T_{u_3})} + \cdots + \log \frac{W(T_{u_k})}{w_x}]/\log(2/(1+\tau)) \\ &= k + \log(W/w_x)/\log(2/(1+\tau)). \end{aligned}$$

We must still add to that the number of links between the lists, which is also bounded by k=$\lfloor \log(W/w_x)/\log(2/(1+\tau))\rfloor$ (see lemma 2). Thus $d'_\tau(x,W) \leq 2k + \log(W/w_x)/\log(2/(1+\tau)) = a_\tau \log(W/w_x)$. where $a_\tau = 3/\log(2/(1+\tau))$. □

To allow update operations, we must store in each node x of the tree an integer between 1 and the outdegree of the node to identify $next[x]$. We furthermore maintain the global value W which is the sum of the weight of the nodes present in the all tree and max_W which is the maximal value of W since the last time that the hotlinks structure was completely rebuilt.

Inserting. We give in this section an algorithm to insert a leaf x of weight $w_x = 1$. The shortest path $x_0, \ldots, x_k$ from the root to the leaf to be inserted is a succession of k hotlinks, heavy tree links, and non-heavy tree links. After finding the shortest path to the parent of the leaf to be inserted, we create the leaf and we check if it is deep, that is, if $k > d'_\tau(x,W)$. If it is, then there must be some node on the path that does not satisfy one of the equations (2) or (4). We then walk up the path verifying those conditions. When walking up from node x_{i+1} to node x_i with $x_{i+1} = next[x_i]$ (heavy tree link) or when $x_{i+1} = jump[x_i]$ (hotlink) we verify eq.(2), and otherwise (non heavy tree link) we verify eq.(4).

To verify condition (2), we must first find the end of the sublist starting at x_i. Let j be the largest integer $< i$ such that $x_{j+1} \neq jump[x_j]$. If $x_{j+1} = next[x_j]$, then the sublist starting at x_i can be constructed by following the next pointers from x_i until the element $jump[x_j]$ is found (see figure 4). Otherwise, x_{j+1} is the head of the heavy path containing x_i, and the sublist starting at x_i can be constructed by following the next pointers until a leaf is reached. Once the path constructed, the weights of the sublists can be computed by exploring exhaustively the subtrees of their elements. To verify condition (4), we explore the subtrees of the children of x_i. Once the scapegoat (node not satisfying one of the conditions) x_i is found, we can consider reconstructing a sublist containing

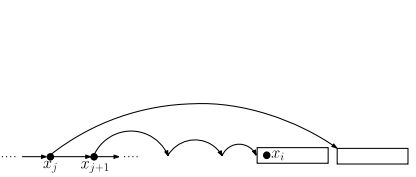

Fig. 4. How to find the end of a sublist

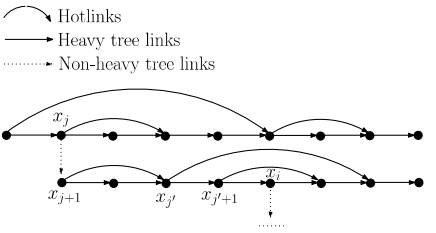

Fig. 5. Determination of the subtree to reconstruct

it. The sublist to reconstruct is the sublist starting at x_i in the first two cases (jumplist violations). We still have to verify that for all next pointers in that sublist, eq.(4) is satisfied. If it is the case, only the jump pointers in that sublist will have to be reassigned. Otherwise, we are in the last case.

For the last case, eq.(4), we will have to reconstruct an entire subtree. Let j be the largest integer $< i$ such that (x_j, x_{j+1}) is a non-heavy tree link, and let j' be the smallest integer $j < j' < i$ such that $x_{j'+1} = next[x_{j'}]$ if it exists, and otherwise $j' = i$. That is, $x_{j'}$ is the first element in the heavy path of x_i for which the next pointer is used, if it exists (See the figure 5). The subtree to reconstruct in this case is the subtree starting at $x_{j'}$. This is to ensure that no jump pointer will point to elements no longer in the same heavy path after the reconstruction. It is easy to see that in this case the weight of the subtree to reconstruct is no more than roughly twice the weight under the scapegoat element. Indeed, we know that the element $jump[x_{j'}]$ is a descendant of the node x_i, thus the jump sublist of $x_{j'}$ has a smaller weight than the weight under the scapegoat element. As the weighted jumplists guarantee a balance between the weight of the jump and the next sublists, we can conclude that the weight of the jumplist defined by $x_{j'}$ (equal to $W(T_{x_{j'}})$) is no more than roughly $2W(T_{x_i})$.

Let C_i be the jump sublist or subtree starting at x_i we want to reconstruct. Before reconstructing it, we have to verify that its reconstruction will not leave deep nodes in its subtree. If $i \leq a_\tau \log(W/W(C_i))$, then we know the reconstruction of the sublist will guarantee that no nodes be deep after the reconstruction, since the length of a search for x in the reconstructed sublist/subtree will be at most $a_\tau \log(W(C_i)/w_x)$. Otherwise, we know there is another scapegoat element higher along the path to the root and we can afford to continue looking for it.

Let us now consider a sequence of insert operations beginning with a tree whose total weight is W, we wish to show that the amortized complexity per insert is $O(\log W)$. We begin by defining a nonnegative *potential function* for the hotlinks tree T. Let $\Phi(x) = \varphi_1 + \varphi_2$ be the potential of the element x. Where φ_1 is the potential function relative to the reassignation of hotlinks in heavy paths (see section 2.2) and φ_2 is the potential function relative to the reassignation of hotlinks in subtree. Thus let $y_1, \ldots, y_l$ be the children of x and let its potential be equal to $\varphi_2(x) = \max(0, \max_i W(T_{y_i}) - W'(T_x)/2)$. Thus a node that indicates the heaviest tree as its next element has a potential of 0, and a node that does

not satisfy the condition eq.(4) has a potential of $\Omega(W'(T_x))$. The potential of the tree is the sum of the potential of its nodes.

It is easy to see that by increasing their cost by only a constant factor, the insertion operations pay for the increase in potential of the nodes. That is, whenever we pass by an element x to insert a new node as a descendant of x, we can pay for the increased potential in x that may be required by the resulting increase in $\Phi(x)$. The potential of the scapegoat node x_i, like all the nodes that do not observe the condition eq.(4), is $\Theta(W'(T_{x_i}))$. Therefore, this potential is sufficient to pay for finding the scapegoat element and reassigning the hotlinks of the sublist or subtree that has to be reconstructed. These operations have complexity $\Theta(size(x_i)) < \Theta(W(x_i))$ (where $size$ is the number of elements in a sublist or a subtree).

Deletion. The deletion operation consists of removing a leaf node x of the tree of weight 1. We first search the node x in the tree then we removed it, and we update $W[C]$. Then, if $W < max\_W/2$, we reassign all the hotlinks of the tree, and we reset max_W to W. This method does not affect the search time t by much: $a_\tau \log(max\_W/w_x) \leq a_\tau(\log(W/w_x) + 1)$.

Since we perform $\Omega(n)$ operations between two successive rebuilds due to delete operations we can pay for them in the amortized sense (with n equal to the number of elements in the tree). Thus for a sequence of delete operations, the amortized complexity per deletion of a leaf node i with the dynamic assignment method is equal to $O(\log(W/w_i))$.

Reweighting. This operation is exactly the same as the insertion and the deletion except that we do not actually insert or delete a node. See section 2.2.

Acknowledgments

The authors thank Pat Morin for many stimulating discussions.

References

1. A.Andersson and T.W.Lai. Fast updating of well-balanced trees. In *Proc. of the Second Scandinavian Worshop on Algorithm Theory*, volume 447 of *LNCS*, pages 111–121, 1990.
2. P. Bose, E. Kranakis, D. Krizanc, M. V. Martin, J. Czyzowicz, A. Pelc, and L. Gasieniec. Strategies for hotlink assignments. In *Proc. 11th Ann. Int. Symp. on Algoritms and Computation (ISAAC 2000)*, volume 1969 of *LNCS*, pages 23–34, 2000.
3. P. Bose, D. Krizanc, S. Langerman, and P. Morin. Asymmetric communication protocols via hotlink assignments. In *Proc. of the 9th Coll. on Structural Information and Communication Complexity (SIROCCO 2002)*, pages 33–40, 2002.
4. H. Brnnimann, F. Cazals, and M. Durand. Randomized jumplists : A jump-and-walk dictionary data structure. In *Proc. 20th Ann. Symp. on Theoretical Aspects of Computer Science (STACS 2003)*, volume 2607 of *LNCS*, pages 283–294, 2003.

5. J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. V. Martin. Evaluation of hotlink assignment heuristics for improving web access. In *Proc. 2nd Int. Conf. on Internet Computing (IC'2001)*, pages 793–799, 2001.
6. M. Drott. Using web server logs to improving site design. In *Proc. ACM Conf. on Internet Computer Documentation*, pages 43–50, 1998.
7. A. Elmasry. Deterministic jumplists. Technical report, DIMACS, 2003.
8. S. Fuhrmann, S. O. Krumke, and H.-C. Wirth. Multiple hotlink assignment. In *Proc. 27th Int. Workshop on Graph-Theoric Concepts in Computer Science*, volume 2204 of *LNCS*, pages 189–200, 2001.
9. I. Galperin and L. Rivest. Scapegoat trees. In *Proc. 4th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 165–174, 1993.
10. O. Gerstel, S. Kutten, R. Matichin, and D. Peleg. Hotlink enhancement algorithms for web directories. In *Proc. 14th Ann. Int. Symp. on Algorithms and Computation (ISAAC 2003)*, volume 2906 of *LNCS*, pages 68–77, 2003.
11. E. Kranakis, D. Krizanc, and M. V. Martin. The hotlink optimizer. In *Proc. 3rd Int. Conf. on Internet Computing (IC'2002)*, pages 33–40, 2002.
12. E. Kranakis, D. Krizanc, and S. Shende. Approximate hotlink assignment. In *Proc. 12th Ann. Int. Symp. on Algoritms and Computation (ISAAC 2001)*, volume 2223 of *LNCS*, pages 756–767, 2001.
13. R. Matichin and D. Peleg. Approximation algorithm for hotlink assignments in web directories. In *Proc. Workshop on Algorithms and Data Structures (WADS 2003)*, volume 2748 of *LNCS*, pages 271–280, 2003.
14. N.Abramson. Information theory and coding. *McGraw Hill*, 1963.
15. M. Perkowitz and O. Etzioni. Towards adaptive Web sites: conceptual framework and case study. *Computer Networks*, 31(11-16):1245–1258, 1999.
16. A. Pessoa, E. Laber, and C. de Souza. Efficient algorithms for the hotlink assignment problem: The worst case search. In *Proc. 15th Ann. Int. Symp. on Algorithms and Computation(ISAAC 2004)*, volume 3341 of *LNCS*, page 778, 2004.
17. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proc. Workshop on Algorithms and Data Structures (WADS 1989)*, volume 382 of *LNCS*, pages 437–449, 1989.
18. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.