Adaptive Website Design using Caching Algorithms

Justin Brickell
Department of Computer Sciences
University of Texas at Austin
Austin, TX, USA
jlbrick@cs.utexas.edu

Inderjit S. Dhillon
Department of Computer Sciences
University of Texas at Austin
Austin, TX, USA
inderjit@cs.utexas.edu

Dharmendra S. Modha
IBM Almaden Research Center
San Jose, CA, USA
dmodha@us.ibm.com

ABSTRACT

Visitors enter a website through a variety of means, including web searches, links from other sites, and personal bookmarks. In some cases the first page loaded satisfies the visitor's needs and no additional navigation is necessary. In other cases, however, the visitor is better served by content located elsewhere on the site found by navigating links. If the path between a user's current location and his eventual goal is circuitous, then the user may never reach that goal or will have to exert considerable effort to reach it. By mining site access logs, we can draw conclusions of the form "users who load page p are likely to later load page q." If there is no direct link from p to q, then it would be advantageous to provide one. The process of providing links to users' eventual goals while skipping over the in-between pages is called *shortcutting.* Existing algorithms for shortcutting require substantial offline training, which make them unable to adapt when access patterns change between training sessions. We present improved online algorithms for shortcut link selection that are based on a novel analogy drawn between shortcutting and caching. In the same way that cache algorithms predict which memory pages will be accessed in the future, our algorithms predict which web pages will be accessed in the future. Our algorithms are very efficient and are able to consider accesses over a long period of time, but give extra weight to recent accesses. Our experiments show significant improvement in the utility of shortcut links selected by our algorithm as compared to those selected by existing algorithms.

Categories and Subject Descriptors

I.5.2 [**Pattern Recognition**]: Design Methodology; H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

Keywords

Data Mining, Adaptive Web Sites, Caching, Pattern Mining, Access Logs, Shortcutting

1. INTRODUCTION

As websites increase in complexity, they run headfirst into a fundamental tradeoff: the more information that is available on the website, the more difficult it is for visitors to pinpoint the specific information that they are looking for. A well-designed website limits the impact of this tradeoff, so that even if the amount of information is increased significantly, locating that information becomes only marginally more difficult. Typically, site designers ease information overload by organizing the site content into a hierarchy of topics, and then providing a navigational tree that allows visitors to descend into the hierarchy and find the information they are looking for. In their paper on adaptive website design [**?**], Perkowitz and Etzioni describe these static, master-designed websites as "fossils cast in HTML." They claim that a site designer's *a priori* expectations for how a site will be used and navigated are likely to inaccurately reflect actual usage patterns, especially as the site adds new content over time. As it is infeasible for even the most dedicated site designer to understand the goals and access patterns of all site visitors, Perkowitz and Etzioni proposed building websites that mine their own access logs in order to automatically determine helpful self-modifications.

One example of a helpful modification is *shortcutting*, in which links are added between unlinked pages in order to allow visitors to reach their intended destinations with fewer clicks. Typically a limit N is imposed on the maximum number of outgoing shortcuts on any one particular page. The shortcutting problem can then be thought of as an optimization problem to choose the N shortcuts per page that minimize the number of clicks needed for future visitors to reach their goal pages. These shortcuts may be modified at any time based on past accesses in order to account for anticipated changes in the access patterns of future visitors. Finding an optimal solution to this problem would require an exact knowledge of the future and a precise way of determining each user's goal. However, shortcutting algorithms must provide shortcuts in an on-line framework, so the shortcuts must be chosen without knowledge of future accesses. Rather than solving the optimization problem exactly, shortcutting algorithms use heuristics and analyze past accesses in order to provide good shortcuts.

In this paper, we draw a novel analogy between shortcutting algorithms, which maintain an active set of shortcuts on

WEBKDD'06, August 20, 2006, Philadelphia, Pennsylvania, USA.

each page, and caching algorithms, which maintain an active set of items in cache. The goal of caching algorithms—maximizing the fraction of future accesses for items in the active set—is analogous to the goal of shortcutting algorithms. The main contribution of this paper is the CacheCut algorithm for shortcutting. By using replacement policies developed for caching applications, CacheCut is able to run with less memory than other shortcutting algorithms, while producing better results. A second contribution is the FrontCache algorithm, which uses the same caching techniques in order to select pages to promote on the front page with direct links.

The remainder of this paper is organized as follows. In Section **??** we discuss related work in adaptive website problems. In Section **??** we give definitions for terms that are used throughout the remainder of the paper. Section **??** gives a formulation of the shortcutting problem and presents two shortcutting algorithms from existing literature. In Section **??** we detail our CacheCut algorithm for shortcutting, and also describe the FrontCache algorithm for promoting pages with links on the front page. Section **??** describes our experimental setup and the results of our experiments. Finally, in Section **??**, we offer some concluding thoughts and suggest directions for future work.

2. RELATED WORK

Perkowitz and Etzioni [?] issued the original challenge to the AI community to build adaptive web sites that learn visitor access patterns from the access log in order to automatically improve their organization and presentation. Their follow-up paper [?] presented several *global* adaptations that affect the presentation of the website to all users. One adaptation from their paper is "index page synthesis," in which new pages are created containing collections of links to related but currently unlinked pages. In his thesis [?], Perkowitz presents the shortcutting problem as a global adaptive problem, in which links are added to each page to ease the browsing experience of all site visitors. Ramakrishnan *et al.* [?] have also done work in global adaptation; they observe that frustrated users who cannot find the content they are looking for are apt to use the "back" button. The authors scan the access log looking for these "backtracks" to identify documents that are misclassified in the site hierarchy, and correct these misclassifications.

Other work has explored adaptations that are *individual*, rather than global; sometimes this is referred to as *personalization.* It is increasingly common for portals to allow users to manually customize portions of their front pages [?]. For instance, a box with local weather information can be provided based on zip code information stored in a client cookie. The Newsjunkie system [?] provides personalized newsfeeds to users based on their news preferences. Personalization is easy when users provide both their identity and their desired customizations, but more difficult when the personalization must take place automatically without explicit management on the part of the user. The research community has made some stabs at the more difficult problem. Anderson and Horvitz [?] automatically generate a personal web page that contains all of the content that the target user visits during a typical day of surfing. Frayling *et al.* [?] improve the "back" button so that it jumps to key pages in the navigation session. Eirinaki and Vazirgiannis [?] give a survey of the use of web mining for personalization.

Operating at a level between global adaptations and individual adaptations are *group* adaptations. The mixture-model variants of the MinPath algorithm [?] are examples of group-based shortcutting algorithms. When suggesting shortcuts to a website visitor, they first classify that visitor based on browsing behavior, and then provide shortcuts that are thought to be useful to that class of visitors. Classifying users requires examining the "trails" or "clickstreams" in the access log, which are the sequences of pages accessed by individual visitors. Other researchers have investigated trails without the intention of adapting a website. Banerjee and Ghosh [?] use trails to cluster users. Cooley *et al.* [?] discover association rules to find correlations such as "60% of clients who accessed page A also accessed page B." Yang *et al.* [?] conduct temporal event prediction, in which they also estimate *when* the client is likely to access B.

Our work follows the global model of shortcutting seen in [?], in which shortcutting is viewed as a global adaptation that adds links to each page that are the same for every visitor. Like Perkowitz' algorithm, when choosing shortcuts for a page p we pay close attention to the number of times other pages q are accessed *after* p within a trail; however, our algorithm provides improvements in the form of reduced memory requirements and higher-quality shortcuts. A related work by Yang and Zhang [?] sought to create an improved replacement policy for website caching by analyzing the access log. In contrast, our work uses existing caching policies to create an improved website.

3. DEFINITIONS

Before describing CacheCut and other shortcutting algorithms from the literature, we provide some definitions that we will use throughout the paper.

Site Graph. The *site graph* of a website with n unique pages is a directed n-node graph $G = (V, E)$ where $e_{pq} \in E$ if and only if there is a link from page p to page q.

Shortcut. A *shortcut* is a directed connection between web pages p and q that were not linked in the original site graph, *i.e.*, $e_{pq} \notin E$.

Shortcut Set. The *shortcut set* S_p of a page p is a set of pages $\{q_1, ..., q_N\}$ such that there is a shortcut from p to each $q_i \in S_p$.

Access Log. The *access log* records all requests for content on the website. Common webservers like Apache produce an access log automatically. In its raw form, the access log contains information that is not needed by shortcutting algorithms. We strip away this unnecessary information and formally consider the access log to be a sequence of tuples of the form

$$\langle \textit{client}, \textit{page}, \textit{time}, \textit{referrer} \rangle,$$

where *client* is the identity of the client accessing the website, *page* is the page requested by the client, and *time* is the time of access. Some shortcutting algorithms also use the *referrer* field, which is the last page the client loaded before loading the current page. This information is self-reported by the client and tends to be unreliable, so we prefer not to use it.

Trail. A *trail* is a sequence of pages $\{p_1, p_2, ..., p_k\}$; we also assume there is a function Time such that $\text{Time}(p_i)$ returns the time at which page p_i was accessed within the trail. A trail represents a single visit to the website by a single client, starting at page p_1 at time $\textsc{Time}(p_1)$and ending at page p_k at time $\textsc{Time}(p_k)$. In order to determine which sequences of page requests constitute a single visit, we require that $\textsc{Time}(p_k) - \textsc{Time}(p_1) < 10\text{min}$. Of course, it is possible to change the 10 minute value.

Note that there need not necessarily be a link in the original site graph between page p_i and page p_{i+1}. This is in contrast to other definitions of trails, which use the referrer field in order to require that the trail be a sequence of clicks. We adapt the more inclusive definition because there are many ways for a user to navigate from p_i to p_{i+1} without following a direct link. For instance, the user could have navigated to an external site with a link to p_{i+1}, typed in the address for p_{i+1} manually, or followed a link on p_{i-1} after using the "back" button. If visitors to page p often visit q later in the session, this is good evidence that a shortcut from p to q would be useful, regardless of how those visitors found their way from p to q.

Trail-Edge. The set of *trail-edges* E_T of a trail T is the set of forward edges in the site graph spanned by T. If $T = \{p_1, p_2, ..., p_k\}$, then $E_T = \bigcup_{i=1}^{k-1} \bigcup_{j=i+1}^{k} e_{ij}$; note that $|E_T| = \binom{|T|}{2}$. In a sense, E is the set of edges that *could be useful* to the client in moving from page p_1 to page p_k. Some of these edges are available as links on the site graph, while others may become available as shortcuts. A user does not need to have all trail-edges available to successfully navigate a trail, but each edge that is available increases the number of ways to navigate from page p_1 to page p_k.

4. SHORTCUTTING

Shortcutting adds links to the site graph that allow users to quickly navigate from their current location to their goal page. If a user on page A wishes to visit page E, he may find that there is no way to navigate to E without first loading intermediate pages B, C, and D. Providing a direct link from A to E would save him 3 clicks. If we transformed the site graph G into a complete graph by adding every possible link, then any user could reach any page in a single click. However, this is an impractical transformation because a human visitor cannot make sense of a webpage with hundreds of thousands of links. This is representative of a general tradeoff that we encounter whenever adding links to pages. Pages become more accessible when they have more inlinks, but become more confusing when they have more outlinks. We typically address this tradeoff by limiting the number of shortcut links per page to N, a small value such as 5 or 10.

With the restriction of N shortcuts per page in place, an optimal shortcutting algorithm is one that chooses the N shortcuts for each page p that minimize the number of clicks required for site visitors to navigate to their goal page. If we could look into the future and read the minds of site visitors, then each time a visitor loaded a page p, we could choose the shortcuts on p based on that visitor's goal. In this case, only a single shortcut is needed for page p—a shortcut to the visitor's goal page Since it is not possible to look into the future, algorithms for shortcut selection must instead mine the web access log for access patterns of past visitors, and then provide shortcuts that would have been helpful to past visitors with the assumption that they will also be useful to visitors in the future.

4.1 Evaluating the quality of a shortcutting algorithm

The goal of shortcutting is to reduce the number of clicks that a visitor must make in order to reach his goal page. The shortcutting algorithm must provide shortcuts to the visitor on-line, without any knowledge of where the visitor will go in the future; at the time of suggestion it is impossible to determine whether any of the provided shortcuts will be useful to the visitor. Once a visitor's trail is complete, however, it is possible to examine the trail in its entirety and evaluate the quality of the shortcuts provided at each page in the trail.

Ideally we could evaluate the quality of shortcuts by comparing the number of clicks needed to reach the goal page both before and after shortcutting. Unfortunately, knowledge of an entire trail is not enough to determine which page was the goal. It is possible that the last page of the trail is the goal page, as is the case when visitors leave the web site after reaching their goals. However, it is also possible for visitors to deliberately load several distinct goal pages during their sessions, or to reach their goals midway through their sessions and then browse aimlessly, or to never reach their goals at all.

The shortcut evaluation used in [?] makes the assumption that the last page in a trail is the goal page, even though this assumption may be incorrect for many trails. Rather than make any such assumption about goal pages, we will simply assume that *any* shortcut that allows a visitor to jump ahead in his trail is useful. Then we evaluate the quality of a shortcutting algorithm for a trail T by determining the fraction of trail-edges available to the visitor as shortcuts or links. Formally, let E_T be the set of trail edges of T, let E be the set of edges of the site graph G, and let S_p be the set of shortcuts on page p at the time that page p was visited. Then a trail edge $e_{pq} \in E_T$ is *available* if either $e_{pq} \in E$ or $q \in S_p$. We define the trail-edge hit ratio of a shortcutting algorithm for a trail T as the fraction of trail-edges that are available:

$$HitRatio(T) = \frac{|\text{available trail-edges of } T|}{|\text{trail-edges of } T|}. \qquad (1)$$

The hit ratio ranges from 0 (if none of the pages in the trail are linked or shortcutted) to 1 (if every trail-edge is provided as either a link or a shortcut). Note that the hit ratio will generally increase as we increase the number of shortcuts per page, N. To evaluate the overall success of a shortcutting algorithm, we take a suitably large access log with many thousands of trails and compute the average trail edge hit ratio:

$$AverageHitRatio(Trails) = \frac{\sum_{T \in Trails} HitRatio(T)}{|Trails|}, \qquad (2)$$

where $Trails$ is the set of trails in the access log.

4.2 Perkowitz' shortcutting algorithm

In [?], Perkowitz gives a simple algorithm that we call $\textsc{PerkowitzShortcut}$ for selecting shortcuts; this algorithm is shown in Algorithm ??. $\textsc{PerkowitzShortcut}$ is periodically run offline to update all of the shortcuts on the website, and these shortcuts remain in place until the next time that an update is performed. For every page p, the algorithm counts the number of times other pages are accessed after p in the same trail, and then it adds shortcuts on p to the N pages most frequently accessed after p. PerkowitzShortcut is simple and intuitive; however, it theoretically requires n^2 memory, which can be prohibitive. In practice, the memory requirements of Perkowitz are closer to $O(n)$ when a sparse representation of the count array C is used.

Because the shortcuts are updated offline and no information is retained from the previous time the update was run, there is a tradeoff when choosing how frequently to update. If the updates are too frequent, then there is inadequate time for the probability distribution to settle. In particular, pages p that are infrequently accessed may have poorly chosen shortcuts (or no shortcuts at all, if the algorithm never sees a session that loads p). If the updates are too infrequent, then the algorithm will be unable to adapt to changes in visitor access patterns. Our algorithm CacheCut presented in Section **??** improves on PerkowitzShortcut by using less memory and providing higher-quality shortcuts.

```
Inputs:
  $G = (V, E)$   The $n \times n$ site graph
  $L$            The access log (divided into trails)
  $N$            The number of shortcuts per page
Output:
  A shortcut set $S_p$ for each page $p$

PerkowitzShortcut($G, L, N$)
  Initialize an $n \times n$ array of counters $C$. $C_{pq}$ represents
    how often users who visit page $p$ later go on to visit page
    $q$.
  For each trail $T$ in the access log, and for each page $p$ in
    $T$, find all pages $q$ that occur after $p$ in $T$. If $e_{pq} \notin E$,
    then increment $C_{pq}$.
  For each page $p$, find the $N$ largest values $C_{pq}$, and select
    these to be the shortcut set $S_p$. Output all shortcut sets.
```

Algorithm 1: A basic shortcutting algorithm for generating shortcuts from the current page to popular destinations

4.3 The MinPath algorithm

The MinPath [**?**] algorithm is a shortcutting algorithm developed to aid wireless devices in navigating complicated websites. Wireless devices benefit from shortcuts more than traditional clients because they have small screens and high latency, so each additional page that must be loaded and scrolled requires substantial effort on the part of the site visitor. Although designed with wireless devices in mind, MinPath is a general purpose shortcutting algorithm that can suggest shortcuts to any type of client.

Unlike PerkowitzShortcut and our algorithm CacheCut, MinPath does not associate shortcuts with each page on the website. Instead, it examines the *trail prefix* $\langle p_1, ..., p_i \rangle$ that has brought a visitor to the current page p_i. Based on the prefix, MinPath returns a set of shortcuts specifically chosen for the individual visitor. This approach requires significantly more computation each time that shortcuts are suggested to visitors, but has the potential to provide shortcuts that are more personalized to the individual visitor.

MinPath works in two stages. In the first stage, which occurs offline, MinPath learns a model of web usage. In the second stage, which occurs offline, MinPath uses its model to estimate the *expected savings* of web pages, where the expected savings of a page q is the estimated probability that the user will visit page q multiplied by the savings in clicks required to navigate from the current page to q. For example, suppose that a user is currently at page p and the web usage model calculates that there is a 0.3 chance of that user visiting page q. If it takes 3 clicks to navigate from p to q ($e.g. p \rightarrow a \rightarrow b \rightarrow q$), then the expected savings is $0.3 \cdot (3 - 1) = 0.6$ because a shortcut from p to q would reduce the number of clicks from 3 to 1. After computing the expected savings for all possible destinations from the current page, MinPath presents the user with N shortcuts having the highest expected savings.

The web usage models learned by MinPath estimate the quantity

$$\Pr(p_i = q | \langle p_0, p_1 ..., p_{i-1} \rangle),$$

which is the probability that a user currently at page p_{i-1} will click on the link to page q given that he has arrived at p_{i-1} by the trail $\langle p_0, p_1 ..., p_{i-1} \rangle$. This probability is 0 if there is no direct link from p_{i-1} to q; otherwise, a probability estimate is learned from observed traffic. In [**?**], the authors present several different techniques for learning the web usage models, including a naïve Bayes mixture model and a Markov model. In the first stage of the MinPath algorithm, sophisticated machine learning techniques are acceptable for the learning stage because this stage occurs offline and its complexity does not affect the amount of time needed generate a set of shortcuts (which is done by evaluating the model in real-time). That being said, the evaluation of the model in the second stage does consume a nontrivial amount of time, especially in comparison to other shortcutting algorithms that only need to look up the shortcuts associated with the current page. The evaluation routine calls for a depth-first traversal of the site graph starting at the current page up to a maximum depth d. When MinPath is called on portions of the site graph with particularly high out-degree this can lead to very long evaluation time. The MinPath authors state that during their tests it took MinPath an average of 0.65 seconds to evaluate the web usage model and return shortcuts each time a visitor loaded a page; if MinPath were deployed on a web server intended to serve tens of thousands of requests per second the server would struggle to keep up.

5. THE CACHECUT ALGORITHM

In the CacheCut algorithm, we associate with each page p a cache C_p of size L which stores web pages q that have been accessed *after* p within a trail. It is not possible to store information about every page accessed after p, so CacheCut must carefully choose which L pages to store in each cache. Our ultimate goal is to select the shortcuts on page p from the contents of cache C_p, so we want to store those pages q which are likely to be accessed after p many times again in the future. When a page q is accessed after p that is *not* currently in C_p, we add it to C_p because it is likely to be accessed again. If C_p is full, then we must select one of its elements to remove and replace with q. We refer to the methodology we use to select the element to be replaced as a *replacement policy.*

The main insight in the CacheCut algorithm is that replacement policies designed for traditional caching problems are well suited as replacement policies for shortcut caches. We can draw an analogy between traditional caching and shortcut caching:

- Users (site visitors) are analogous to processes.
- Web pages are analogous to pages in memory.
- The shortcut set is analogous to a cache.

Replacement policies for traditional caching applications are heuristics that attempt to throw out the item that is least likely to be accessed in the future, so that the fraction of future accesses that are for objects currently residing in cache is maximized. If we substitute the traditional caching terms for their shortcutting analogs, we see that the goal for cache replacement heuristics is *identical* to the goal for shortcut replacement heuristics, because we want to maximize the fraction of accesses that come after p that are for pages currently in C_p.

Cache replacement policies are evaluated based on their *hit ratio*, which is the fraction of total accesses that are for objects that were in the cache at the time of access. Put in shortcutting terminology, the hit ratio for a trail T with trail-edges E_T becomes:

$$HitRatio(T) = \frac{|\{e_{pq} \in E_T | q \in C_p\}|}{|E_T|}.$$

If we think of the cache C_p as containing the shortcuts for page p in addition to a permanent set of the original links on page p, then this is identical to the evaluation equation for shortcutting algorithms given in equation (**??**).

5.1 Batched caching

The simplest way of using a caching algorithm to select shortcuts would be to have the shortcut set for page p directly correspond to the cache C_p for page p. To implement a shortcutting algorithm in this way we would set the cache size L equal to the number of shortcuts N, and each time a page $q \in C_p$ was replaced with a page r, we would immediately replace the shortcut from p to q with a shortcut from p to r. When evaluated based on hit ratio this scheme performs well, but it is impractical as a deployed shortcutting scheme because the shortcut set changes too frequently Each visitor who passes through page p updates the cache C_p with every subsequent page access in the same trail. If there are thousands of site visitors, then the caches may update very frequently, which would be confusing to a visitor expecting the shortcuts to remain the same when he refreshes the page.

Our solution is to *not* have the shortcut sets and the caches be in direct correspondence. We update the cache C_p as usual with every in-trail access that occurs after p. However, instead of immediately updating the shortcuts on p, they are left alone. Periodically (say, once every 2 hours) the contents of C_p become the shortcuts on p. This method allows us to continue using unmodified out-of-the-box cache replacement policies, while relieving site visitors from the annoyance of having the shortcut set change too frequently.

5.2 Increasing the size of the underlying cache

Once we have decided to not have the cache C_p and the set of shortcuts on p in direct correspondence, we are freed from the restriction that they need to be the same size. By allowing the cache size L to be greater than the number of shortcuts N, we may keep track of data (such as hit count) about more than N items, which enables a more intelligent choice of shortcuts. If $L = N$, then any page accessed immediately before the periodic update of shortcuts will become a shortcut for the next time period, even if it's a rarely accessed page. With $L > N$, we can exclude such a page in favor of a page that is more frequently accessed.

Allowing $L > N$ is beneficial, but it adds the additional challenge of choosing which N of the L pages in C_p will become the shortcuts on page p. A simple selection policy that performs well in practice is to maintain a hit count for each item in C_p, and then to choose the N items most frequently accessed during the previous time period. The hit counts are reset each time period, so this selection criteria is based entirely on popularity during the previous time period.

In order to expand the selection criteria to consider accesses during all past time periods, we introduce *α-history selection.* The α-history selection scheme has a parameter $0 \le \alpha < 1$; higher α means that less emphasis is placed on recent popularity, and more emphasis is placed on total past popularity. The scheme works as follows: for a page $q \in C_p$, let $A_p(q)$ be the number of times page q was accessed after page p within a trail during the previous time period. Let $H_p(q)$ be the historical "score" of page q in the shortcut set S_p. Initially, $H_p(q) = 0$ for all $q \in S_p$. At the end of each time period when selecting new shortcuts, first update the scores as:

$$H_p(q) = \begin{cases} \alpha H_p(q) + (1-\alpha)A_p(q) & \text{for } q \in C_p \\ 0 & \text{for } q \notin C_p \end{cases}.$$

Now when choosing the shortcuts for page p, we pick the top N pages from C_p using the H_p scores. The α-history selection scheme allows us to consider the popularity of pages in past time periods, but exponentially dampens the influence of the old hits based on their age. Note that $H_p(q)$ and $A_p(q)$ information is discarded the moment that a page q is replaced in cache C_p; this ensures that memory usage is still proportional to the cache size when α-history selection is used. As an additional enhancement, we can weight the A_i values by the total number of hits in time period i so that hits that occur during unpopular times (nighttime) are not dominated by hits that occurred earlier during popular times (daytime). All of our experiments use this enhancement. Note that the PerkowitzShortcut algorithm is equivalent to setting $L = n$, the number of web pages, and $\alpha = 0$.

5.3 CacheCut implementation

The CacheCut algorithm, presented in Algorithms **??** and **??** makes use of the following subroutines:

- Cache(Page p). Returns the cache associated with page p.
- RecordAccess(Cache C, page p, time t). Informs the cache C of a request for page p at time t. Page p is then placed in the cache C, and it is the responsibility of C's replacement policy to remove an item if C is already at capacity. The time t is used by some replacement policies, such as least recently used, to determine which item should be replaced.
- SetHits(Cache C, page p, int x). For a page p assumed to be in cache C, sets the hit count to x.
- GetHits(Cache C, page p). If p is currently in cache C, returns the hit count of p. Otherwise, returns 0.
- SetScore(Cache C, page p, float x). For a page p assumed to be in cache C, sets the score to x.

```
Inputs:
  $G$   The site graph
  $T$   Observed trail $\langle p_0, ..., p_k \rangle$
UpdateTrailCaches($G, T$)
for $i = 0$ to $k - 1$ do
    $C_{p_i} \leftarrow$ Cache($p_i$)
    for $j = i$ to $k$ do
        if there is no link in $G$ from $p_i$ to $p_j$ then
            RecordAccess($C_{p_i}, p_j$, Time($p_j$))
            $hits$ = GetHits($C_{p_i}, p_j$)
            SetHits($C_{p_i}, p_j, hits + 1$)
        end if
    end for
end for
```

Algorithm 2: This routine is called each time a trail completes in the access log, in order to update the caches with the new information from that trail.

- GetScore(Cache C, page p). If p is currently in cache C, returns the score of p. Otherwise, returns 0.

When CacheCut is initialized, every page is associated with an empty cache. As visitors complete trails, the caches of pages along the trails are modified; this takes place in the UpdateTrailCaches routine given in Algorithm **??**. For each trail-edge e_{pq}, two actions are taken. First, the cache C_p is informed of a hit on page q, and the replacement policy chooses an element of C_p to replace with q. Second, a hit count for page q in cache C_p is incremented. Some replacement policies may maintain their own hit counts, but this hit count is used for the α-selection scoring.

Although the caches update with the completion of every trail, the shortcut sets are not updated until a call is made to UpdateShortcuts, which is shown in Algorithm **??**. UpdateShortcuts is periodically called in order to choose the shortcut sets S_p from the caches C_p. This is done using α-selection scoring, as described in Section **??**. The hit counts for each page are reset each time that UpdateShortcuts is called, but some information about prior hit counts is retained in the score.

```
Inputs:
  $P$   Set of web pages $\{p_1, .., p_n\}$
  $N$   Number of shortcuts per page
  $\alpha$   History weighting parameter
UpdateShortcuts($P, N, \alpha$)
for all pages $p$ in $P$ do
    $C_p \leftarrow$ Cache($p$)
    for all pages $q$ in $C_p$ do
        $newScore \leftarrow \alpha \cdot$ GetScore($C_p, q$) $+ (1 - \alpha) \cdot$
            GetHits($C_p, q$)
        SetScore($C_p, q, newScore$)
        SetHits($C_p, q, 0$)
    end for
    $S \leftarrow$ top $N$ pages in $C_p$ by GetScore($C_p, q$)
    Set shortcuts of page $p$ to be $S$
end for
```

Algorithm 3: This routine is run periodically to choose the shortcuts for page p_i from the cache C_i. The shortcuts are scored by a combination of their previous score and their number of recent hits, and the top N are chosen.

5.4 Promoting Pages on the Front Page

The front page of a website serves a special role as a portal to the website's entire content. It is a common point of first entry for visitors to the website, and it provides links and other navigational devices to help visitors access *all* content found on the website. A well-designed front page directs visitors to popular content, so that new visitors who are unfamiliar with the website can quickly find the most useful content pages.

The CacheCut algorithm provides a set of shortcuts for every page including the front page. However, it does not treat the front page differently than other pages, so the shortcuts for the front page are chosen considering only pages that are accessed after the front page within a trail. Because of the front page's special role, it is useful to select a more inclusive set of shortcuts for the front page. Our goal for selecting shortcuts for the front page will be to maximize the total fraction of all page accesses that are for pages linked from the front page.

In order to select shortcuts for the front page we created an algorithm FrontCache that is a modification of the CacheCut algorithm. It maintains a single cache which is updated whenever *any* web page in the web site is loaded—no attention is paid to trails, and in fact there is no need to determine the identity of the client loading the page. Every few hours shortcuts are selected for the front page from this cache. Since only a single cache is maintained, there is no need to save on memory, and it is appropriate to use cache sizes much larger than the size of the shortcut set. We found, for instance, that a cache of size 80 or more is useful when choosing 10 shortcuts. To choose which items in the cache become shortcuts, α-history selection is used, as in CacheCut.

6. EXPERIMENTAL RESULTS

Experimental setup and implementation details

For our experiments, we collected web access log data spanning April 17 to May 16, 2005 from the University of Texas Computer Sciences department website, which has about 120,000 unique web pages. The access log originally included requests for data such as images, movies, and dynamic scripts (asp, jsp, cgi). These data are often loaded as a component to a page rather than as an individual page, which confuses trail analysis because loading a single page can cause multiple sequential requests (such as for the page itself and its 3 images). To address this issue, we removed all requests for content other than html pages, text documents, and Adobe Acrobat documents.

Approximately one-third of the page requests came from automated non-human visitors. Because these robots and spiders access the website in a manner distinct from human visitors, we were able to identify and remove their accesses.

Since trails of length 1 and 2 cannot be improved (assuming that the length 2 trail spans a static link), we restricted our dataset to include only trails of length 3 or greater. Once we performed all the various data cleaning steps, we were left with 89,086 trails with an average length of 7.81 pages. MinPath requires a separate training and testing set, and the access transactions in the training set should logically occur before the transactions in the test set, so when evaluating MinPath we made the first two-thirds of the log training data and the last one-third test data. The other algorithms train for their future shortcuts at the same time that they are evaluating their present shortcuts, so we were able to use the entire access log as test data.

The MinPath algorithm has several parameters; in order to simplify our testing we used a fixed configuration, varying only the number of shortcuts produced. We used the "unconditional model" of web site usage because it was easy to implement (our attempt to obtain the MinPath software from the authors of [?] was unsuccessful). This model had the worst performance of those presented in [?], but its performance was within 20% of the best model, so we feel that it gives a good understanding of the capabilities of MinPath. We did not group pages together by URL hierarchy (effectively making the URL usage threshold equal to 0%), but our training data had 900,000 page requests, (more than 7 times larger than in [?]), which increased the number of pages with accurate usage estimations.

6.1 Choosing the best parameters for CacheCut

The CacheCut algorithm has several different parameters that can affect performance. They are:

- N, the number of shortcuts per page.
- L, the size of the underlying cache.
- alg, the underlying cache replacement policy.
- α, the history preference parameter.

In this section we examine the tradeoffs allowed by each of these parameters, and investigate how they affect the performance of the CacheCut algorithm as evaluated by the formula given in equation (**??**).

The number of shortcuts per page.

In terms of *AverageHitRatio*, it is always beneficial to add more shortcuts. As discussed in the introduction, if we allow N to become arbitrarily large, then *AverageHitRatio* will be 1 since every trail-edge will either be a link or a shortcut. A site designer who wishes to use CacheCut for automated shortcutting will need to choose a value for N that provides a good tradeoff between shortcutting performance, and the link clutter on each webpage. Figure **??** shows how performance increases with the number of shortcuts. The relationship appears to be slightly less than logarithmic.

The size of the underlying cache.

As the size of the underlying cache L increases from N to $2N$, the performance increases substantially. This is because the algorithm is able to retain information about more good pages; when $L = N$ if the algorithm has N good pages in the cache C_p, it is forced to replace one of them with a bad page q when q is accessed after p, and then it loses all of its accumulated data about the good page. The gain as L continues to increase beyond $2N$ is marginal, and it appears that there is little reason to increase L beyond $5N$. Figure **??** shows how performance varies with the size of the cache.

Underlying cache replacement policies.

We tested the following four cache replacement policies:

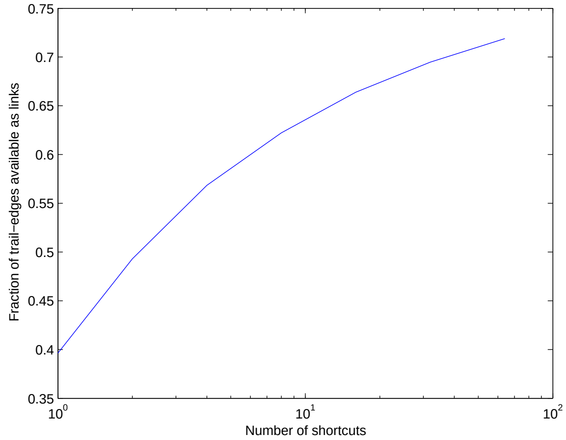

Figure 1: *AverageHitRatio vs. Number of shortcuts.* The more shortcuts that are added to each page, the more likely it is that a trail-edge will be available as a link or shortcut. Here GDF is used with $L = 80$ and $\alpha = 0.9$. With no shortcuts (original links only) the hit ratio is 0.23.

- Least Recently Used (LRU)
  Least Recently Used is a classic cache replacement policy. We associate each item in the cache with its last time of access. The item that was accessed least recently is chosen for replacement.
- Least Frequently Used (LFU)
  Least Frequently Used is another classic cache replacement policy. We associate each item in the cache with the number of times it has been accessed since being placed in the cache. The item that was accessed the fewest times is chosen for replacement.
- Adaptive Replacement Cache (ARC)
  The Adaptive Replacement Cache policy is a recent algorithm described in detail in [?]. The basic idea is to maintain two lists, L_1 and L_2. L_1 maintains pages that have been seen only once recently, while L_2 maintains pages that have been seen at least twice recently. The algorithm is therefore able to capture both recency and frequency. If there are c pages cached in total, then k (where $k \leq c$) are kept in L_1 and $c - k$ are kept in L_2. The parameter k changes with time.
- Greedy Dual Size Frequency (GDF)
  The Greedy Dual Size Frequency policy is described in detail in [?]. It is a state-of-the-art caching algorithm for web proxies that considers recency and frequency, as well as the size of the page. For our application we only store links to pages, so we ignore the page size because it is irrelevant. GDF is a synthesis between LRU and LFU. Like those policies, it keeps cached pages in a priority queue, but it keys the pages by a function of their recency and frequency.

As seen in Figure **??**, the greatest variation in performance between cache replacement policies occurs when L is equal to N, or only slightly larger than N. GDF has the best performance, and LRU has the worst. The poor performance

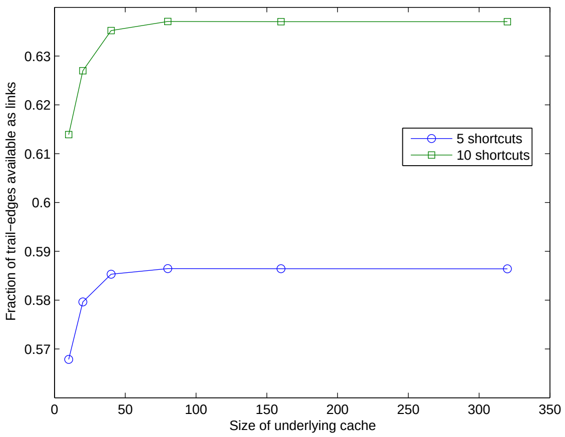

Figure 2: *AverageHitRatio vs. Size of cache.* Increasing the cache size beyond the size of the shortcut set increases the fraction of trail-edges available as shortcuts. Here GDF is used with $\alpha = 0.9$.

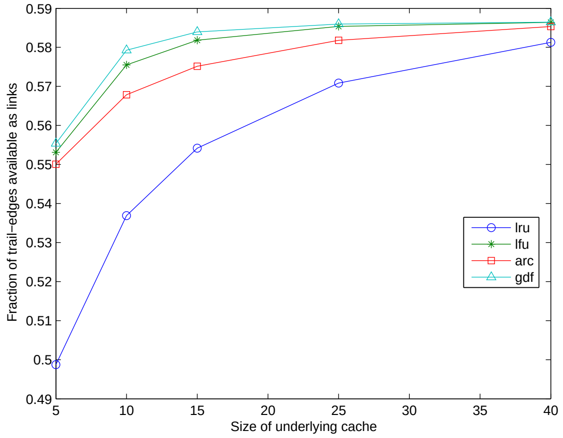

Figure 3: *AverageHitRatio vs. Replacement policy.* GDF is the best replacement policy, but for large values of L all four policies perform nearly identically. Here $\alpha = 0.9$.

of LRU is explained by its lack of consideration of frequency. Because traffic patterns are fairly consistent over time, it's important not to throw out pages that were accessed very often in the past in favor of pages that have been accessed recently, but only infrequently.

The differences between the other three replacement policies are very slight, and become negligible as L increases. Regardless of what replacement policy is used for the cache, the same scoring system is used to choose which N cache elements become shortcuts. As L becomes large, it grows increasingly likely that the N top-scoring pages will be present in all caches, no matter which replacement policy is used. If enough memory is available to support a large L, it would probably be best to use LFU because it is extremely efficient and performs just as well as the more elaborate GDF and ARC policies.

The history preference parameter.

When choosing the shortcuts on a page p, the history preference parameter α allows us to choose what weight will be given to recent accesses A_p and historic popularity H_p. For a detailed discussion, see Section **??**. If $\alpha = 0$, then the score is entirely determined by the recent popularity $H_p(q)$. In this case CacheCut behaves in the same way as PerkowitzShortcut, except that it keeps track of counts for a subset of pages rather than for all pages. If $\alpha = 1$ then the algorithm fails to work properly, because the scores will always be equal to 0 since the history, which is initialized to 0, will never be updated. Figure **??** shows how the value of α affects the quality of shortcuts. Values of α very close to 1 do very well, which suggests that usage patterns on the website are somewhat consistent over time.

6.2 Comparing CacheCut to other shortcutting algorithms

We ran several tests to compare CacheCut to PerkowitzShortcut, MinPath, and a baseline algorithm that selects N shortcuts for each page entirely at random. In the comparisons we chose the best parameters for CacheCut that we found in the previous experiments. They were $L = 80$, the GDF replacement policy, and $\alpha = 0.9$. For PerkowitzShortcut we allowed the time between updates to be quite large, 72 hrs, because doing so produced the best results. Since MinPath requires separate training and testing phases, we partitioned the data as two-thirds training set and one-third test set.

We evaluate the performance of the 4 shortcutting algorithms using three different criteria:

- **Fraction of trails with at least one useful shortcut.** If a site visitor encounters a single shortcut to a desired destination, then the the shortcutting algorithm was useful to that visitor. Among all trails of length 3 or greater, we find the fraction that have at least one trail-edge available as a shortcut. This comparison is presented in figure **??**, and CacheCut clearly outperforms the other algorithms. Adding random shortcuts is barely more useful than having no shortcuts at all, which is not surprising for a website with 120,000 nodes. It is also noteworthy that with only 5 shortcuts per page, about 75% of visitors that *can* have their trail enhanced (*i.e.*, those with trail lengths of 3 or greater) are provided a useful shortcut by the CacheCut algorithm.

- **Average fraction of trail-edges available as shortcuts or links.** This is the *AverageHitRatio* criteria motivated in this paper. The results are given in figure **??**, and once again CacheCut shows the best performance.

- **Average trail length after shortcutting.** In [**?**] when introducing MinPath, the authors state that the goal of their algorithm is to reduce the number of clicks required for a visitor to get from their initial page to their goal page. Because there is no accurate way of determining which page in a trail is the goal page, they assume that the last page in the trail is the goal because after loading it the visitor left the site. Suppose that a visitor's access to the web site is a trail

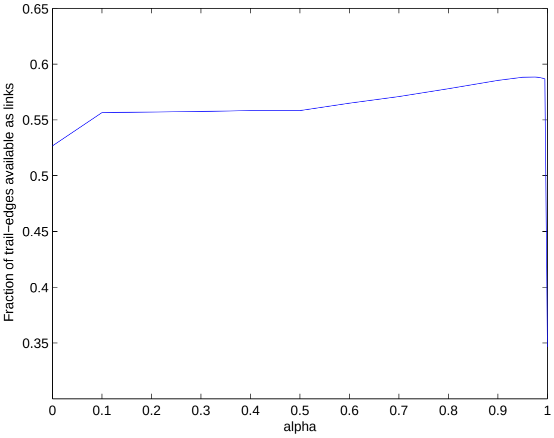

Figure 4: *AverageHitRatio vs. α.* Increasing α to nearly 1 increases performance, but performance drops off precipitously when α is too close to 1. Here GDF is used with $L = 20$ and $N = 5$.

of length $t \geq 3$ pages. By applying shortcutting, we could reduce this length to as few as 2 pages (if there was a direct shortcut from the first page to the last). To determine the length of trails after shortcutting, we assume that at each page, users choose whatever available shortcut leads furthest along their trail; if there are no trail-edges available as shortcuts then the user goes to the next page in their original trail.

If a trail has a loop, then evaluating with this scheme may inappropriately attribute a decrease in trail length to the shortcuts. For instance, if a trail of length 7 has page p as both its second page and its seventh page, then regardless of the shortcutting algorithm used we would report the length after shortcutting as 2 for a "savings" of 5 pages. To get around this problem, we removed from consideration all trails with loops. The average trail lengths for the different shortcutting algorithms are shown in figure **??**. Even though MinPath was designed to minimize this value, it is still outperformed by CacheCut.

6.2.1 Timing tests

The amount of time that shortcutting algorithms require is just as important as the quality of shortcuts, because a shortcutting algorithm that requires too much time will be impractical to deploy. In order to compare runtimes, we used each algorithm to generate 5 shortcuts per page on the portion of the access log used to test MinPath; this portion of the log had 29,249 trails and 1,267,921 trail-edges. The results are shown in Table **??**.

PerkowitzShortcut requires less time than CacheCut, which is understandable because CacheCut must perform a cache replacement for each trail-edge, whereas PerkowitzShortcut needs only to increment a count in its array. MinPath required substantially more time than the other two algorithms, because it must evaluate its web-usage model each time a visitor loads a page in order to determine what shortcuts to suggest. The amount of time required to evaluate the model is related to the out-degree of the current page and its offspring, and is very long compared to the constant-time lookup the other algorithms need to suggest shortcuts.

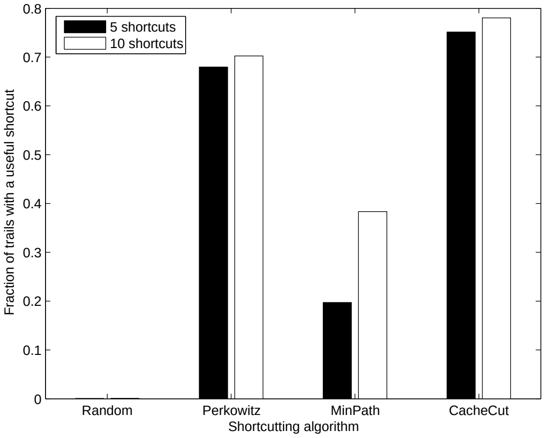

Figure 5: *Trails aided by shortcutting.* The fraction of trails that have at least one trail-edge available as a shortcut.

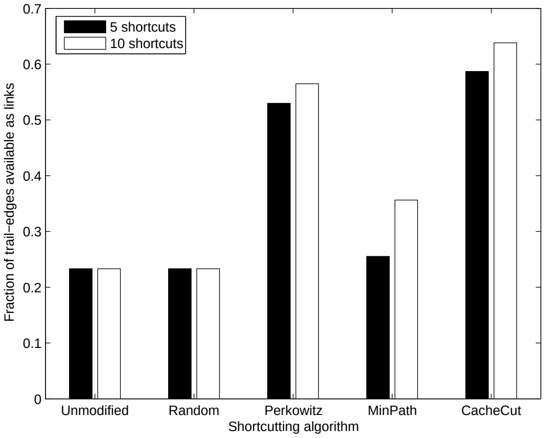

Figure 6: *Edges available as shortcuts.* The fraction of trail edges available as shortcuts, averaged over all trails.

7. CONCLUSIONS AND FUTURE WORK

Shortcutting algorithms add direct links between pages in order to reduce the navigational effort necessary for site visitors to find their desired content. If there was some way to know with absolute certainty what each user's goal page was, then it would be easy to provide a link to that page. Since this is not possible, we instead add a handful of links to each page p that a user is likely to find useful.

In this paper, we introduced the CacheCut shortcutting algorithm, which uses the predictive power of cache replacement policies to provide website shortcuts that are likely to be useful to site visitors. Compared to other shortcutting algorithms in the literature, CacheCut is fast and resource efficient. CacheCut can process a month's worth of access

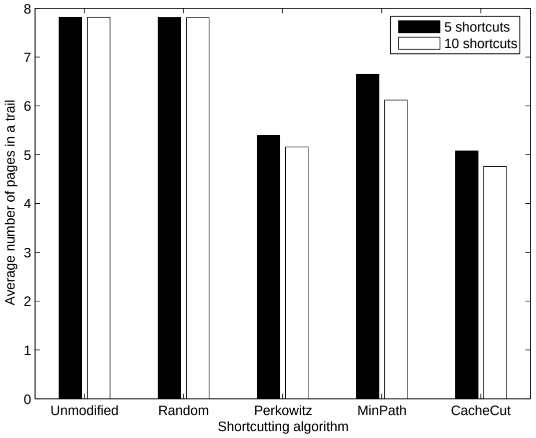

Figure 7: *Length of trails after shortcutting.* The average length from the first node in a trail to the last, when the visitor is assumed to take the available shortcut at each node that leads furthest along the trail.

MinPath	CacheCut	Perkowitz
8.15 hours	32 seconds	17 seconds

Table 1: Time to generate 5 shortcuts per page on 29,249 trails

logs in a few seconds, so it is suitable for real-time deployment without straining the webserver.

The CacheCut algorithm is seen to be very effective despite its simplicity. Most visitor trails that can be improved by shortcutting are improved by CacheCut, and in fact a significant fraction of trail-edges are available as shortcuts for the average trail. More complicated shortcutting algorithms such as MinPath consider a visitor's entire trail rather than only the current page, but this added complexity does not improve the quality of shortcuts provided, and the additional computation needed makes them impractical to deploy. Compared to PerkowitzShortcut, our algorithm produces higher-quality shortcuts and is guaranteed to need only $O(n)$ memory. In some applications this guarantee may be desirable, even though PerkowitzShortcut uses $O(n)$ memory in practice.

In the future, it would be useful to deploy a shortcutting algorithm on an active website, and observe how it influences the browsing behavior of visitors. Analyzing the performance of shortcutting algorithms offline as done in this paper means that we must ignore the possibility that visitors' browsing trails would be different in the presence of shortcutting links. A deployed version would also need to keep track of how often the presented shortcuts are used, and retain the most utilized shortcuts rather than replace them with new shortcuts. Deploying a shortcutting algorithm requires determining exactly how shortcuts will be added to webpages as links. One possibility is to modify the web server so that when it serves a web page to a visitor, HTML code for the shortcuts are automatically added to the page. The downside of this approach is that it may be difficult to find an appropriate place within the page to add the shortcuts so as to not ruin the page formatting.

Acknowledgments

We would like to thank Albert Chen for his contribution to a preliminary version of this work. This research was supported by NSF grant CCF-0431257, NSF Career Award ACI-0093404, and NSF-ITR award IIS-0325116.

8. REFERENCES

[1] C. R. Anderson, P. Domingos, and D. S. Weld. Adaptive web navigation for wireless devices. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 2001.

[2] C. R. Anderson and E. Horvitz. Web montage: A dynamic personalized start page. In *WWW '02: Proceedings of the eleventh international conference on World Wide Web*, pages 704–712. ACM Press, 2002.

[3] A. Banerjee and J. Ghosh. Clickstream clustering using weighted longest common subsequences. In *Proc. of the Workshop on Web Mining, SIAM Conference on Data Mining*, pages 33–40, 2001.

[4] L. Cherkasova. Improving www proxies performance with greedy-dual-size-frequency caching policy. *HP Laboratories Report No. HPL-98-69R1*, 1998.

[5] R. Cooley, B. Mobasher, and J. Srivastava. Web mining: Information and pattern discovery on the world wide web. In *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*. IEEE, November 1997.

[6] M. Eirinaki and M. Vazirgiannis. Web mining for web personalization. *ACM Trans. Inter. Tech.*, 3(1):1–27, 2003.

[7] E. Gabrilovich, S. Dumais, and E. Horvitz. Newsjunkie: Providing personalized newsfeeds via analysis of information novelty. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 482–490. ACM Press, 2004.

[8] N. Megiddo and D. S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[9] N. Milic-Frayling, R. Jones, K. Rodden, G. Smyth, A. Blackwell, and R. Sommerer. Smartback: Supporting users in back navigation. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 63–71. ACM Press, 2004.

[10] M. Perkowitz. *Adaptive Web Sites: Cluster Mining and Conceptual Clustering for Index Page Synthesis*. PhD thesis, University of Washington, 2001.

[11] M. Perkowitz and O. Etzioni. Adaptive web sites: an ai challenge. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

[12] M. Perkowitz and O. Etzioni. Towards adaptive web sites: Conceptual framework and case study. *Artificial Intelligence*, 118([1-2]):245–275, 2000.

[13] R. Srikant and Y. Yang. Mining web logs to improve website organization. In *WWW '01: Proceedings of the tenth international conference on World Wide Web*, pages 430–437. ACM Press, 2001.

[14] Yahoo!, Inc. My Yahoo! `http://my.yahoo.com`.

[15] Q. Yang, H. Wang, and W. Zhang. Web-log mining for quantitative temporal-event prediction. *IEEE Computational Intelligence Bulletin*, 1(1):10–18, 2002.

[16] Q. Yang and H. H. Zhang. Web-log mining for predictive web caching. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1050–1053, 2003.