Document downloaded from:

http://hdl.handle.net/10251/43243

This paper must be cited as:

De La Ossa Perez, BA.; Gil Salinas, JA.; Sahuquillo Borrás, J.; Pont Sanjuan, A. (2013). Referrer Graph: A cost-effective algorithm and pruning method for predicting web accesses. Computer Communications. 36(8):881-894. doi:10.1016/j.comcom.2013.02.005.

The final publication is available at

http://dx.doi.org/10.1016/j.comcom.2013.02.005

Copyright Elsevier

Referrer Graph: a cost-effective algorithm and pruning method for predicting web accesses

B. de la Ossa, A. Pont, J. Sahuquillo and J. A. Gil
Department of Computer Engineering, Universidad Politecnica de Valencia
Camino de Vera, s/n, 46022 Valencia (Spain)
berospe@doctor.upv.es, {apont, jsahuqui, jagil}@disca.upv.es

October 7, 2014

Abstract

This paper presents the Referrer Graph (RG) web prediction algorithm and a pruning method for the associated graph as a low-cost solution to predict next web users accesses. RG is aimed at being used in a real web system with prefetching capabilities without degrading its performance. The algorithm learns from users accesses and builds a Markov model. These kinds of algorithms use the sequence of the user accesses to make predictions. Unlike previous Markov model based proposals, the RG algorithm differentiates dependencies in objects of the same page from objects of different pages by using the object URI and the referrer in each request. Although its design permits us to build a simple data structure that is easier to handle and, consequently, needs lower computational cost in comparison with other algorithms, a pruning mechanism has been devised to avoid the continuous growing of this data structure. Results show that, compared with the best prediction algorithms proposed in the open literature, the RG algorithm achieves similar precision values and page latency savings but requiring much less computational and memory resources. Furthermore, when pruning is applied, additional and notable resource consumption savings can be achieved without degrading original performance. In order to reduce further the resource consumption, a mechanism to prune de graph has been devised, which reduces resource consumption of the baseline system without degrading the latency savings.

1 Introduction

Currently, the web is being massively used, thus constantly increasing the traffic in the network as well as the load that servers manage. Although nowadays web users have higher bandwidth connections, they still perceive huge latencies when navigating the web due to overloaded elements (e.g., network, servers, switches, or intermediate hardware) and long message transference times. Consequently,

the reduction of the latency perceived by users when browsing the web is still a crucial research issue.

To reduce this latency, techniques like caching, geographical replication and prefetching have been proposed and used. Nowadays, caching techniques are widely implemented since they achieve important latency savings. Big companies usually implement web replication by using CDNs to reduce their websites access time, but this solution is expensive and many companies and organizations cannot afford it. Web prefetching techniques are orthogonal to caching and replication techniques, so that they can be applied together to achieve a better web performance.

Web prefetching can highly reduce web latency, as many research works appeared in the open literature [1, 2, 3, 4] show. In a previous work [5] we concluded that page latency savings achieved by web prefetching could range from 39% to 52% in real conditions. However, this technique has been rarely implemented in commercial products due to three main reasons. The first one is that early proposals of web prefetching required modifications in the standard protocols [6], but these are no longer needed since prefetching is compatible with the current standards and software. The second reason was that the effectiveness of web prefetching was initially related with high bandwidth requirements [7], which was a problem in the early proposals that has disappeared in current communication scenarios. Finally, the third reason was that the most efficient prediction algorithms proposed require a lot of computational resources in order to generate precise predictions because they work by capturing previous user accesses to build a model of user patterns to predict future accesses [8].

The first two drawbacks mentioned above prevented the massive adoption of web prefetching and have been properly addressed over the last years. The design of more attractive web prefetching proposals dealing with resource consumption is critical to encourage the widespread use of web prefetching. This paper focuses on designing a prediction algorithm with low computational cost, able to accurately predict the next user accesses, without adding extra service time that could negatively affect the quality of web services.

This paper presents the Referrer Graph (RG) prediction algorithm as a low-cost algorithm that achieves precision values and web latency savings similar to the ones of the best proposals that can be found in the open literature. The proposed RG prediction algorithm aims to be a low-cost alternative to the existing algorithms. RG learns from access patterns using the URI and the Referrer of web requests to predict future accesses. Unlike previous Markov model based proposals, the RG algorithm differentiates dependencies in objects of the same page from objects of different pages by using the object URI and referrer in each request. The resource consumption, the page latency savings and the traffic overhead are evaluated by means of recent, real and representative traces. Results show that RG achieves page latency savings which are similar to the ones of the other evaluated prediction algorithms with lower memory and computational resource consumption.

Although the RG prediction algorithm achieves good performance with low resource consumption as shown in [9], this cost increases continuously over time

due to the continuous learning process, as it happens in any other prediction algorithm proposed in the literature. This paper extends the work presented in [9] by devising a prune mechanism in order to further reduce the increase in resource usage. In this context, Section 6 has been included to describe and evaluate how the proposed mechanism noticeably reduces computational resources while sustaining the performance. This mechanism, referred to as pruning algorithm, removes from the graph, both periodically and continuously, those nodes, arcs, and occurrences that lose their value over time. In this way, RG can run continuously during unbounded periods of time.

The remaining of this paper is organized as follows. Section 2 presents the motivation and previous related work. Section 3 describes the RG prediction algorithm. Section 4 describes the evaluation methodology. Section 5 shows and analyzes the experimental results Section 6 presents and evaluates the pruning algorithm. Finally, Section 7 presents some concluding remarks.

2 Motivation and Related Work

Prediction algorithms can be classified in two main groups according to the type of information used to make predictions [3].

The first group includes algorithms that predict future accesses based on the previous access patterns. Two subgroups can be distinguished: one that consists of algorithms that use Markov models [1, 2, 4, 10] and the other with algorithms that use data mining techniques [11, 12, 13]. Many prediction algorithms based on Markov models can be found in the literature, and some of them provide high precision predictions but at the expense of intensive computation and memory consumption. The resource consumption of data mining based algorithms is even higher.

The second group contains the algorithms that analyze the web content to make predictions. Some authors propose to combine the analysis of the content with usage profiles [14], others apply neural networks to keywords extracted from HTML content [15], and some others detect similarities in context words around links in the HTML content [16]. Proposals [15, 16] are based on the object popularity and the associated hyperlinks, but they do not consider the relationship among objects.

Regarding the prediction algorithms based on Markov models, one of the most widely used in the literature is the Dependency Graph (DG) proposed by Padmanabhan and Mogul [1]. DG builds a dependency graph that depicts the pattern of accesses to the objects, where there is a node in the graph for each object that has ever been accessed. Other well-known algorithm is the Prediction by Partial Match (PPM), proposed by Palpanas and Mendelzon [2], which also uses a Markov model to store the context of accesses. Both algorithms achieve a high precision in the predictions, but unfortunately this fact is not directly related to high latency savings for web users because the algorithms do not consider the structure of the current websites [4].

The Double Dependency Graph (DDG) algorithm, proposed by Domènech

et al. [4], is based on DG but it considers the structure of current websites by differentiating between pages and embedded objects. DDG provides useful predictions to effectively reduce the user perceived latency when downloading web pages.

The prediction algorithms mentioned above try to learn the user patterns from the sequence of accesses. These algorithms consider that two objects are related if they are requested by the same user closely in time.

Other Markov algorithms learn user patterns from the site structure. Zukerman et al. [8] compares different prediction models, some of them consider the order in which documents are requested, and others the structure of the server site. But the study does not present any algorithm in detail and no results about the actual page latency savings. Zhu et al. [10] proposed to build a Markov model from web log files and use it to make predictions. This work mainly concentrates on the compression of the transition probability matrix. It does not present any algorithm in detail, and does not study the performance of the prediction algorithm.

Using current prediction algorithms in the real world presents some problems with the resource consumption. Several works have addressed this topic in order to reduce the complexity of the prediction models and, consequently, their computational and memory costs. In this sense, Deshpande et al. [17] focuses on pruning Markov models of web prediction algorithms. They present several techniques to combine Markov models of different order to reduce the state-space complexity while maintaining the prediction accuracy. They also propose three schemes for pruning states: states with low frequency of occurrence; states with low confidence on state outgoing transition; and states with high error associated with a state (determined by using offline verification with a trace). They concluded that the Markov models after pruning achieve similar or better accuracy than the original models.

3 Referrer Graph

3.1 General Description

The Referrer Graph (RG) prediction algorithm builds a graph based on client requests. Each requested web object is represented by a node in the graph. For each web request that reports its referrer, an arc is created from the referred node (predecessor) to the requested nodes (successor). The resulting graph is used to make predictions that produce hints which are returned to the web client.

The idea of a prediction algorithm that considers the site structure accessed by users is motivated by the fact that users commonly navigate following hyperlinks on web pages. This model considers that two objects are related to each other if there is a hyperlink between them. Hence, the model considers structural information about the website instead of the sequence of the requested objects. In general, the algorithms that use the sequence of requests to build

the graph need a window of last accesses for each client session in order to establish arcs among the previous accesses and the current one. RG does not need this window because this algorithm keeps the relationship between objects based on their direct reference, which is known thanks to the referrer field in the request [1]. Each HTTP object request includes information about the requested object and its referrer, so in order to establish an arc it is not necessary to keep track of previous accesses of the same client session.

RG handles each web request independently of the context of the whole navigation session. Therefore, RG does not need to keep track of each user navigation session. On the contrary, prediction algorithms that use the sequence of accesses identify each navigation session by using the IP address of the client. This is a problem when several web sessions use the same IP address, which happens when several clients use the same proxy server, or when they connect from the same local network masked by a single IP address. Therefore, the performance of RG is not negatively affected by any of these circumstances.

DDG is a prediction algorithm based on web access patterns while RG is based on web structure [18]. This is the main reason why the cost of RG is lower than that of previous proposals. The graph structure built by RG is also a double dependency graph. However, in the case of RG this structure is lighter due to the use of the referrer information instead of the sequence of accesses. This simple way of constructing a lighter graph allows RG to obtain similar or better benefits with much less cost than DDG and the other evaluated algorithms.

3.2 Data Structures

RG builds a first-order Markov model, since only the previous access is used to define the context of the current request. The graph is directed, loopless, and cyclic.

A Markov model is a directed cyclic graph where the next node only depends on the current state. The node represents the context of the user, that is, the recent accesses. Nodes are connected by arcs, which represent the transition between contexts. The arc weight represents the transition confidence of moving from the predecessor node to the successor node. The *order* of a Markov model indicates how many past accesses are used to define the context in a node.

In the proposed model, a node represents a web object, and it is identified by the object URI. A directed arc from a predecessor to a successor node represents the relationship between both nodes.

To illustrate several working aspects of the RG algorithm, let's assume a Calendar website, as shown in Figure 1. The main page (*Calendars*) is linked to some other secondary pages that focus on different types of calendars: there is a different page for each year calendar (*C2007*, *C2008* and *C2009*); there is page with the rules used to build the Gregorian calendar (*Gregorian*), another

[1] Note that in the original HTTP specification document, the header field was misspelled as "referer", instead of the correct English word "referrer".

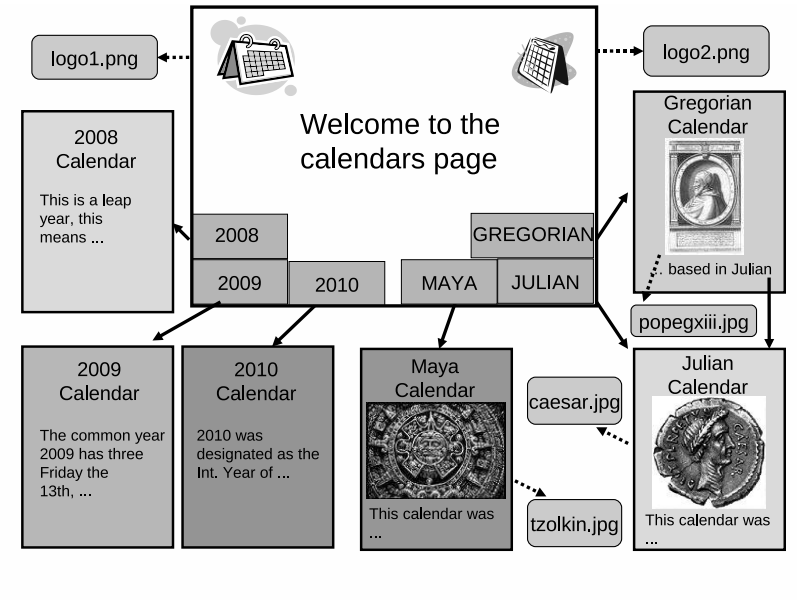


Figure 1: Example website

page with the rules for the Mayan calendar (*Mayan*), and a third one with the rules for the Julian calendar (*Julian*). Images and logos are embedded objects of their own pages.

Figure 2 shows an example graph built by RG for a simple navigation performed in the Calendar website. In the graph there is a primary arc that has node *Calendars*as predecessor, and node *Julian*as successor. This means that a user requested the object represented by node *Julian*, and that request indicated as referrer the object represented by node *Calendars*.

Initially, the graph is empty. Then is built and updated through a learning process. We define the occurrence of a node as the number of requests to the represented object, and the occurrence of an arc as the number of requests to

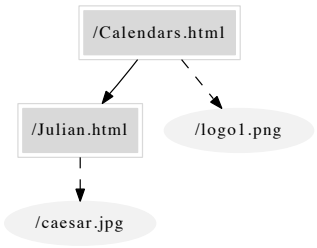


Figure 2: Simple graph with two primary and two secondary nodes

the successor node which provided as referrer the predecessor node. For each web request, if the graph already contains a node representing that object, the node occurrence is increased. Otherwise, the node is created with occurrence set to one. Also, an arc is created or, if it already exists, its occurrence is increased. To avoid the uncontrolled growth of the graph, it can be periodically pruned by removing those nodes and arcs that become less representative.

Nodes and arcs can be classified in two main types: primary and secondary. Primary nodes represent objects that are requested explicitly by users, while secondary nodes represent embedded objects that are requested by the web client, not by the user. A secondary node is not predecessor of any arc and its object's MIME type corresponds to a typical embedded object (image, video, script, style sheet, etc). A secondary node is promoted to primary when an arc having that node as predecessor is established. Thus, it is not possible that an arc has as predecessor a secondary node. On the contrary opposite, primary nodes can never be promoted back to secondary.

An arc inherits the type of its successor node. When a secondary node is promoted to primary, the arcs that have this node as successor are also promoted to primary.

A web page visited by a user consists of a main object and several embedded objects. The graph represents this page as a primary node, several secondary nodes, a primary arc from the referrer node to the primary node, and secondary arcs from the primary node to the secondary nodes.

In the proposed data structure, the graph stores for each node: the object URI, node type, node occurrence, list of referrers, list of primary arcs, and list of secondary arcs. The data stored for an arc is the destination URI, arc occurrence, and arc transition confidence.

A secondary node keeps a list of referrer nodes because this allows to quickly find them if the node is promoted to primary.

3.3 Learning Process

The learning part of RG is summarized in Figure 3 and consists of three main steps: updating the requested node, updating the arc from the referrer node to the requested node, and promoting the referrer node, if required.

When an object is requested, its corresponding node is found, and the occurrence and the list of referrers of the node are updated. If the node has not been created yet, it is created with a type according to the object MIME type, and inserted in the graph. If this information is not available, or the MIME type does not allow to discern the node type, then it is built as a secondary node.

If the node is secondary, it adds the referrer URI to a list. This is useful in case of node promotion, as it helps to quickly find the nodes that link to this node and the arcs that have this node as successor.

If the request provides as referrer the requested object, or if the referrer is a node that does not exist in the graph (that includes the case of external

referrers), then the learning process ends. This ensures that arcs have a different node as predecessor and successor.

If the referrer already has a node in the graph and there is an arc from the referrer to the requested object, the arc occurrence is increased. In addition, all the arcs that have the referrer as the predecessor node are updated: the arc transition confidence is calculated as the arc occurrence divided by the predecessor node occurrence. If the predecessor is a secondary node, it is promoted to primary and all the arcs with that node as successor are promoted to primary, too.

3.4 Prediction Process

Figure 4 shows the main steps of the algorithm that makes predictions and provides hints. When a user requests an object, the corresponding node is looked for in the graph. If the node does not exist or it is secondary, no prediction is made and no hints are provided. On the contrary, if the node exists and is primary, all its primary arcs are analyzed. Those arcs that have a transition confidence greater than a given threshold are included in a list of primary arcs, ordered by transition confidence. Then the successor nodes of such arcs are looked for and their secondary arcs are analyzed. Those secondary arcs with a transition confidence (multiplied by the previous primary arc transition confidence) greater than a secondary threshold are included in a list of secondary result arcs, also ordered by arc transition confidence.

The lists of primary and secondary arcs are then concatenated. The URIs associated to these arcs are the hints that will be provided as predictions. The definitive list of hints can be cut to provide only the first N hints.

3.5 Working Example

This section presents an example of a set of clients sessions browsing the Calendar website, the graph built by the learning process, and the hints provided by the prediction algorithm.

Table 1 lists the web requests performed during the client session, which are processed by the learning process of RG. Figure 5 shows the corresponding graph. The primary and secondary nodes are depicted with solid and dashed lines, respectively. Nodes are labeled with the URI and their occurrence while arcs are labeled with the arc occurrence and transition confidence. Table 2 shows the hints that RG would provide using the graph previously shown if a client performed different requests. The main threshold and the secondary threshold are not enforced in this example.

Figure 3: Algorithm for learning from user access and building the RG graph

Input:
  rg: RG graph
  uri: URI requested
  $referrer$: Referrer provided in the request
  $mime$: MIME type of the requested object
Output:
  rg: RG graph with improved knowledge
{ **UPDATING NODE: Updating the requested node** b }
$b \leftarrow$ find node with $uri \in rg$ or build new node, with type based on $mime$
b occurrence $\leftarrow b\ occurrence + 1$
if b type = secondary **then**
  Add to b list of referrers: $referrer$
end if
Store b in rg
{ **UPDATING ARC: Updating arc** $\vec{ab}$ }
if $uri = referrer$ or $referrer \notin rg$ **then**
  return rg
end if
$a \leftarrow$ find node with $referrer \in rg$
$\vec{ab} \leftarrow$ find arc with $uri \in a$ or build new arc
$\vec{ab}$ occurrence $\leftarrow \vec{ab}$ occurrence +1
Store $\vec{ab}$ in a
for all arcs $\vec{at} \in a$ in rg **do**
  $\vec{at}$ transition confidence $\leftarrow \vec{at}$ occurrence / a occurrence
end for
{ **PROMOTION: Promoting Referrer node** a }
if a type = secondary **then**
  a type $\leftarrow$ primary
  for all uris $urix \in a$ list of referrers in rg **do**
    $x \leftarrow$ find node with $urix \in rg$
    $\vec{xa} \leftarrow$ find secondary arc with $referrer \in x$
    x secondary arcs $\leftarrow$ remove $\vec{xa}$ from x secondary arcs
    x primary arcs $\leftarrow x$ primary arcs $\bigcup \vec{xa}$
    Store x in rg
  end for
end if
Store a in rg
return rg

Table 1: Web requests in example client session

URI requested	**URI of referrer**
/Calendars.html	-
/logo1.png	/Calendars.html
/logo2.png	/Calendars.html
/Gregorian.html	/Calendars.html
/popegxiii.jpg	/Gregorian.html
/Julian.html	/Gregorian.html
/caesar.jpg	/Julian.html
/Calendars.html	-
/logo1.png	/Calendars.html
/Maya.html	/Calendars.html
/tzolkin.jpg	/Maya.html
/Calendars.html	www.search.com
/logo1.png	/Calendars.html
/Gregorian.html	/Calendars.html
/popegxiii.jpg	/Gregorian.html
/Julian.html	/Gregorian.html
/caesar.jpg	/Julian.html
/Calendars.html	/Julian.html

Table 2: Hints provided if client requests URI

URI	**Hints URI and probability**
/Calendars.html	/Gregorian.html 50%, /Maya.html 25%, /popegxiii.jpg 50%, /tzolkin.png 25%
/Gregorian.html	/Julian.html 100%, /caesar.jpg 100%
/Julian.html	/Calendars.html 50%, /logo1.png 37.5%, /logo2.png 12.5%
/Maya.html	-

Figure 4: Algorithm for giving hints based on RG graph

```
1: Input:
2:   rg: RG graph
3:   u: last user access
4:   th: primary threshold
5:   thsec: secondary threshold
6: Output:
7:   h: Set of hints
8: for all output primary arcs a ∈ u in rg do
9:    if a transition confidence ≥ th then
10:      hptemp ← hptemp ⋃ {a}
11:      for all output secondary arcs e ∈ a in rg do
12:         if e transition confidence  · a transition confidence ≥ thsec then
13:            hstemp ← hstemp ⋃ {e}
14:         end if
15:      end for
16:   end if
17: end for
18: hpsorted ← sort hptemp by higher probability
19: hsuniq ← delete duplicates in hstemp
20: hssorted ← sort hsuniq by higher probability
21: h ← hpsorted ⋃ hssorted
22: return h
```

4 Evaluation Methodology

4.1 Simulation Framework

To carry out the experiments in order to show the improvements of the RG algorithm, we use a pool of simulated clients with prefetching capabilities fed by real web traces. These clients perform requests to a simulated web server implementing different prediction algorithms, according to the architecture described and implemented in [19]. The server sends the predictions as hints to the clients for them to prefetch.

Several assumptions have been taken into account to carry out the experiments. First, the prefetching engine (located at the clients) prefetches the hints during the browser idle time, like Mozilla-based browsers. Second, each client can set up two HTTP 1.1 non-pipelined persistent connections to the web server, which is the maximum number of connections that the standard recommends. In the prefetching phase, only one of this connections is used (like Mozilla-based browsers). Third, experiments assume an infinite cache in the clients. The main reason of this assumption is that we use real traces to feed the clients, that is, experiments reply those original GET requests made by the clients obtaining a 200 OK response. Therefore, we have no information about the original browser cache hits.

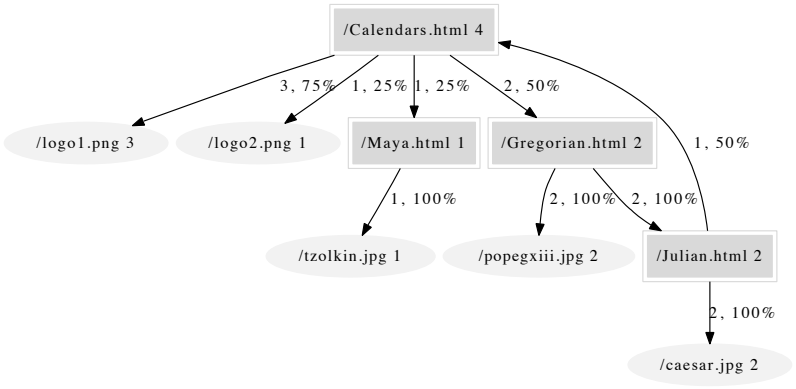


Figure 5: Graph of RG after some simple navigation sessions

4.2 Workload Description

The experiments were performed using two recent web traces (A and B) from different websites. The trace files were logged by Apache 2 web servers in a custom format that included, among other information, the referrer, the user agent, and the object latency. Table 3 summarizes the main characteristics of these traces. As observed, some characteristics widely vary between traces because the corresponding websites serve different contents to distinct user profiles. This fact will noticeably impact on the performance achieved by web prefetching, as shown in Section 5.

Trace A was obtained from the dynamic website of a dynamic website that acts as the home page of an open source project, which mainly contains news posts, documentation, and forums. Most of the content is generated dynamically by PHP scripts that query a database. However, the dynamic URIs are persistent since they are written using an Apache 2 feature that does not use URI query strings.

Trace B is from a school website which mainly consists of news posts, and information addressed to students, staff, and visitors. The content of this site is not dynamically generated and its visitor community is much more constrained.

A web session is a group of page requests made by the same web browser (identified by its IP address). The trace files do not indicate when a session finishes, so we assume that a browser idle time longer than 15 minutes represents the end of the web session.

4.3 Performance Indexes

The page latency saving has been used as the main performance index for measuring prediction and prefetching effectiveness in this work because our aim is to study the maximum benefit perceived by web users.

Table 3: Trace characteristics

Characteristic	**Trace A**	**Trace B**
Starting date	Sep, 27 2007	Mar, 21 2005
Ending date	Jun, 18 2008	Nov, 22 2006
Unique IP addresses	131 668	21 566
Browsing sessions	317 268	127 186
Page requests	992 037	884 249
Average page latency (seconds)	0.877 ± 0.117	1.314 ± 0.240
Unique objects	6790	1168
Object requests	5 654 371	2 244 916
Requests of objects smaller than 10 kB	77%	70%
Bytes transferred (MB)	28 135	35 779

The page latency saving percentage ($\nabla PL(\%)$) has been calculated as:

$$\nabla PL(\%) = (1 - \frac{\text{Average PL with Prefetching}}{\text{Average PL without Prefetching}}) * 100$$

The page latency is defined as the elapsed time from when the user demands a page until the web browser receives all the objects in the page.

All performance indexes have been obtained with a 95% confidence interval. Nevertheless, for the sake of clarity, only average values are shown in the figures, as the interval lengths are always less than 5% of the average value. The run length for each single experiment is 120K user requests and each run consists of 20 intervals equally sized. For each single experiment a preliminary warming-up phase of 70 intervals (420 000 object requests) is carried out before collecting statistics.

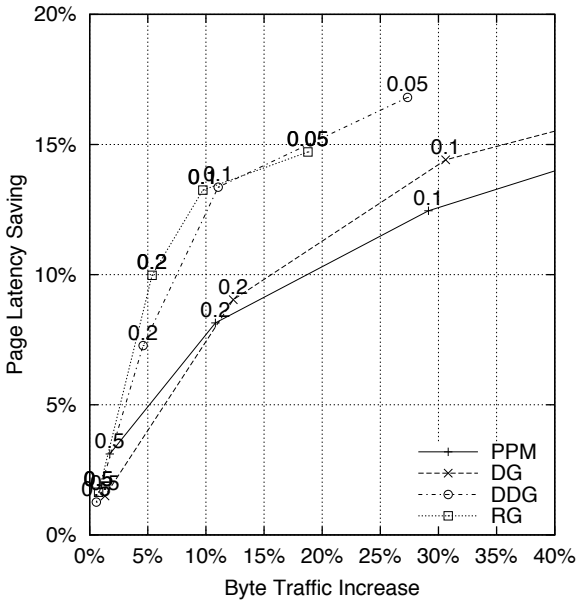
5 Experimental Results

This section compares the cost-benefit and the resource consumption of the RG prediction algorithm against other three well-known algorithms. These algorithms are the Prediction by Partial Match (PPM) [2], the Dependency Graph (DG) [1], and the Double Dependency Graph (DDG) [4].

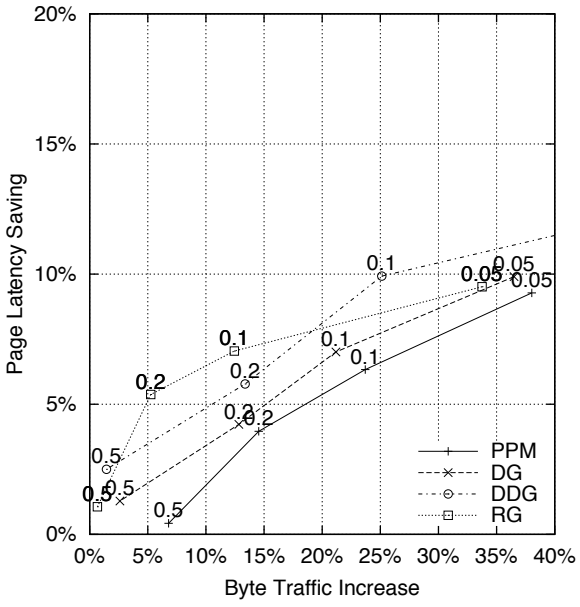
The algorithms parameters were individually tuned according to their intrinsic characteristics in order to obtain the best cost-benefit ratio with the less resource consumption. We found that, in both traces, the best configuration was the use of a lookahead window size of 2 in both DG and DDG, and a first-order Markov model in PPM.

5.1 Page Latency Saving and Byte Traffic Increase

Figure 6 illustrates the cost and benefit obtained by the studied prediction algorithms. The page latency saving quantifies the benefit, while the byte traffic



(a) Trace A



(b) Trace B

Figure 6: Page latency saving versus byte traffic increase

increase measures the incurred cost. Experiments were run for different values of the prediction algorithms threshold parameter, because this is the most appropriate parameter to modify the aggressiveness of the algorithm prediction. Only those hints with a higher probability than the threshold are returned to the web client. The numbers inside the figure show the threshold values used in the experiment run.

DDG and RG obtained better cost-benefit ratio than PPM and DG. This difference is specially noticeable in Trace A (see Fig. 6(a)), but it can also be seen in Trace B (see Fig. 6(b)). This difference arises because DDG and RG discern among primary and secondary objects in the graph; thus, they can provide more useful hints. As observed, RG is the preferable algorithm under low-cost constrains, while DDG is the best when no constrains are imposed.

5.2 Resource Consumption

To increase the prediction accuracy, the prediction algorithms store a lot of information about user's navigation and, consequently, have to perform a deeper information analysis handling a high number of variables to make predictions. As a consequence, prediction algorithms become more and more complex. In other words, the algorithms require more computational and memory resources both to learn from the user behaviour and to make predictions. Therefore, the research must concentrate not only on the precision of the prediction algorithms, but also on their resource consumption. A prediction algorithm whose resource consumption increases exponentially is not appropriate for real usage. When two prediction algorithms provide similar precision, it is preferable the one with lower resource consumption.

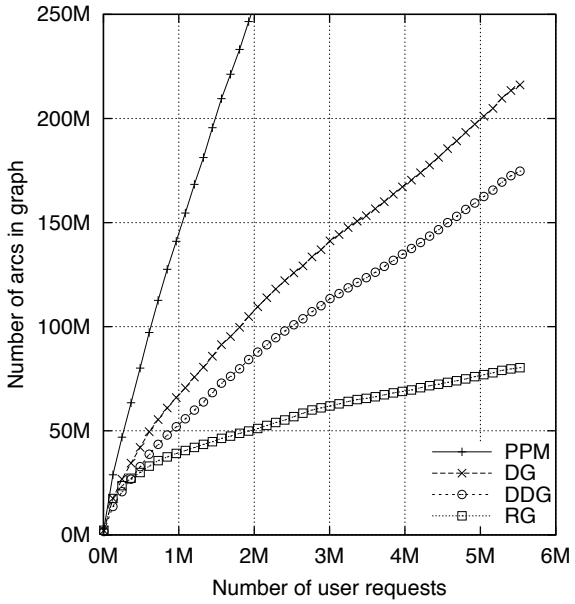
We quantified the resource consumption of the previously mentioned algorithms. To quantify the memory consumption we measured the number of arcs in the graph that is built by each prediction algorithm. The number of nodes in the graphs built by the algorithms DG, DDG and RG is identical because all the algorithms receive the same number of object requests, and all of them create a node for each requested object. As an approximation to the computational consumption, we measured the service time required by the algorithms to make predictions.

The algorithms DG, DDG, and RG create the same nodes in their graphs, although they are connected in a different way. However, PPM builds a data structure that is completely different. Consequently, its data structure is not directly comparable to the other algorithms, as each requested object is represented with one or more nodes in the PPM tree.

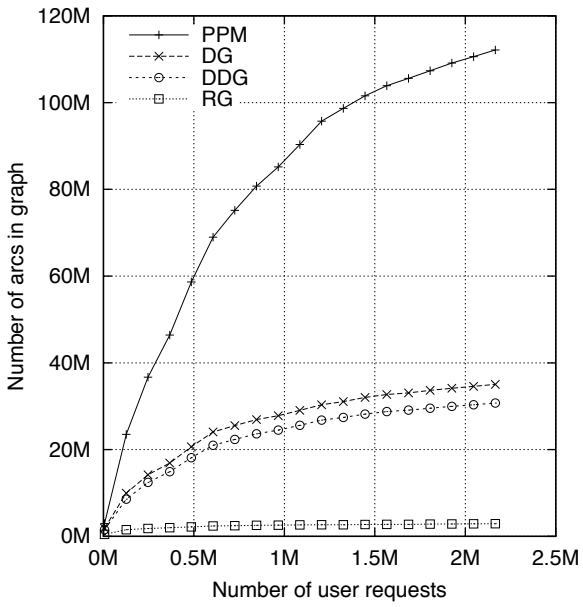
Figure 7 shows the number of arcs in the graph of each algorithm during the experiment measured in number of user requests.

The data structure created by any algorithm strongly depends on the website navigation tree. This is the cause of the big difference in cost between Trace A and Trace B.

The algorithm that always created fewer arcs in its graph was RG. The reason is that RG creates an arc between two nodes when a web request of the



(a) Trace A



(b) Trace B

Figure 7: Number of arcs in graph when threshold is 0.1

second node references the first one. This is the main difference between RG and the other algorithms, where an arc is created between two nodes when they are requested sequentially during a short lapse of time.

Figure 8 depicts the prediction service time during the experiment, that is, the time consumed by the implementation of the prediction algorithm to make a prediction. This time consists of two main components: the learning phase time and the prediction phase time. A server equipped with a 64-bit Intel Xeon Processor 3.2GHz, 2M Cache, 4 cores, with 4 GB of RAM was used to perform the experiments.

As one can observe, the prediction service time (Fig. 8) is directly related to the number of arcs in the graph (Fig. 7). The reason is that those algorithms consume most of their prediction time looping over the arcs in the graph and performing actions with each arc.

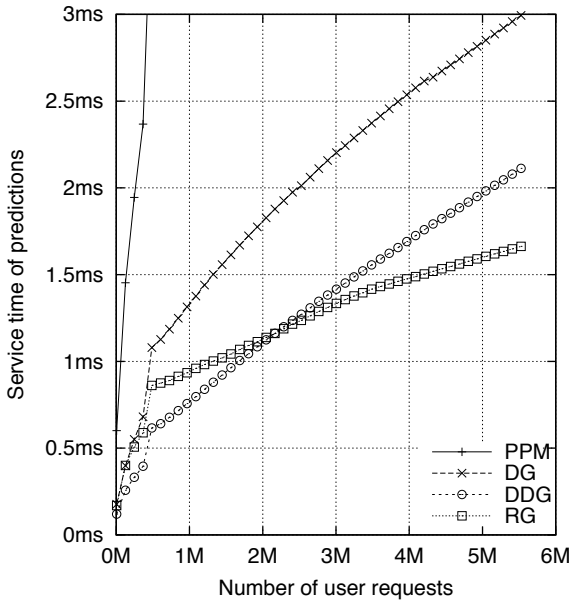
RG was the implemented algorithm that required less time to make predictions, followed by DDG, DG and PPM. However, in all cases the service time increased as the experiment advanced. The reason is that the traces included requests for several months. During the months when the trace was captured, new pages were added to the website. Therefore, the prediction algorithms were continuously adding more nodes to the graph, as well as arcs between old and new nodes. As a consequence, the prediction service time increased. To avoid the continuous increase of this time, we propose to prune the graph, that is, to reduce the graph complexity, as described below.

6 Graph Pruning

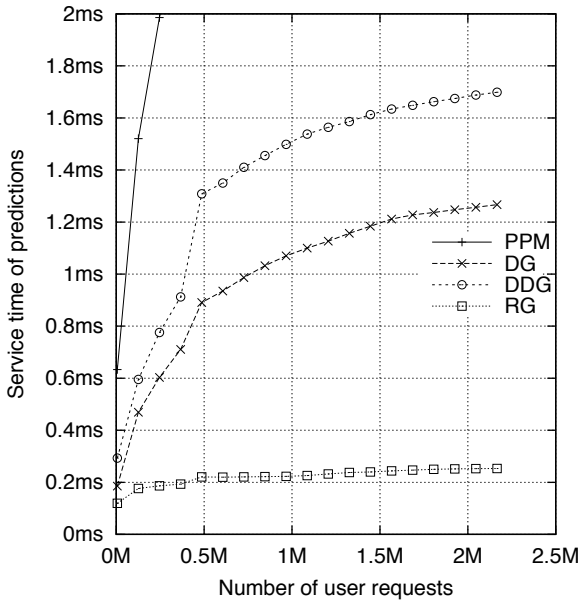
This section presents a method for pruning the graph built by RG. First, the main questions related to the growing of the graph and data structures associated with the prediction algorithm are discussed. Then, the proposed pruning algorithm is presented, and the design choices are explained in detail. An example of RG pruning illustrates how the algorithm works in practice. Finally, the benefits of RG pruning against the baseline RG are evaluated.

6.1 General Issues

The graph associated with a prediction algorithm is dynamically built over time and updated with the corresponding information when a user access is performed. As a result, the graph grows and continually needs more computational resources. The problem arises when some information stored in the graph becomes stale or useless. This fact can happen due to two main reasons: i) the changes on web navigation patterns, and ii) the removal of some website pages. The first case appears because users' interests vary over time. This may happen depending, among others, on the kind of contents the site provides (e.g., sports news, financial news, program documentation, ...). The second case occurs because, when some pages are removed from a website, some knowledge in the graph becomes useless since users will no longer visit such pages. Furthermore,



(a) Trace A



(b) Trace B

Figure 8: Prediction service time when threshold is 0.1

keeping obsolete and useless information in the graph not only wastes memory and computational resources in the prediction engine, but also has a negative impact on the performance because the precision can noticeably decrease. That is, if such information was used by the prediction algorithm, it could provide useless hints. The process of pruning a graph consists of walking on the graph to check which parts have become obsolete, and therefore suitable to be removed.

The design of a pruning algorithm must address two major concerns: resource consumption, and prediction accuracy. The first one means that pruning must not increase the resource consumption of the original prediction algorithm. That is, the sum of resource consumption due to the prediction and the pruning algorithms must not exceed the consumption when pruning is not performed. This entails that the pruning algorithm itself must be a low-consumption process and removing information results in a simpler graph that needs less computational resources. The second concern means that pruning must not have adverse impact on the effectiveness of the web prediction. In other words, pruning the graph implies the removal of information, so if useful information is pruned, prediction accuracy could have adverse effects.

6.2 Proposed Pruning Algorithm

The graph built by RG has two elements that grow over time: nodes that represent pages, and arcs that represent transitions between pages. Node and arc occurrences are counters that increase continuously, as they represent the number of times a page or a transition have been observed. According to the implementation of the data structures, the action of pruning a node also prunes the arcs having that node as predecessor. On the other hand, occurrences are internally represented with integer values, so an integer overflow can raise arithmetic exceptions because an arc or a node is being highly accessed over time. In this sense, an effective pruning could help to avoid such exceptions.

The design of a pruning algorithm is determined by two main decisions: i) what to prune, and ii) when to prune.

The first decision consists in selecting those elements that must be pruned from the graph. This process must cover the pruning of the three main elements of the graph: nodes, arcs, and occurrence values. Notice that pruning nodes and arcs means to remove them from the graph, while pruning occurrences refers decreasing their value. In order to perform a fair pruning, decreasing the occurrence values must keep the overall consistency, that is, these values must be kept proportional to each other. To this end, the occurrences of the arcs are all reduced at the same time, then the occurrence of the predecessor nodes is reduced in the same ratio.

The second decision refers to the points in time at which pruning is performed. This can be done either continuously, periodically, or both. In a continuous pruning, when an element of the graph is accessed, only that element and the directly related elements are checked for pruning. This process is performed dynamically and synchronously with the learning process, so computational consumption extends over time. On the other hand, in a periodic

pruning all the graph contents are checked for pruning at fixed intervals. In the latter case, the computational consumption concentrates on the lapse of time when pruning is triggered.

Figure 9 summarizes the pruning process devised for the RG algorithm. Continuous pruning is used for arcs, and periodical pruning for nodes. Notice that the pruning and learning algorithms (Figure 3) must be interleaved as follows: first, updating the requested node and the arc from its referrer; second, pruning arcs and reducing occurrences; then, promoting the referrer node; and finally pruning nodes. Consequently, the order in the complete algorithm will include the following steps:

1. UPDATING NODE (Figure 3)
2. UPDATING ARC (Figure 3)
3. PRUNING ARCS AND REDUCING OCCURRENCES (Figure 9)
4. PROMOTION (Figure 3)
5. PRUNING NODES (Figure 9)

The criterion used for pruning arcs and occurrences, as highlighted in Figure 9 (PRUNING ARCS AND REDUCING OCCURRENCES), is the following. When a request indicates as its referrer a node that already exists in the graph, and the node occurrence exceeds a threshold (*node_occ_th*), its outgoing arcs are checked for pruning. An arc is pruned when its transition confidence is lower than the primary *threshold*; otherwise the arc occurrence value is reduced by a factor (*occ_reduction_factor*). As mentioned in Section 3.3, the arc transition confidence is calculated as the ratio of its occurrence value to the occurrence value of its predecessor node. Proceeding in this way, the occurrence value estimates the arc popularity. Once all the arcs leaving a node have been updated, the occurrence of the node is also reduced by the mentioned factor to maintain this popularity.

Node pruning is highlighted in Figure 9 (PRUNING NODES). A global *access counter* is used to determine the point in time at which the pruning of all nodes in the graph must start. This counter increases its value each time a web request occurs. When this counter surpasses a given value (*pruning_th*), the node pruning process starts. Nodes are pruned when they i) do not reach a minimum popularity or ii) have not been accessed for a long time. A node is considered popular enough to be maintained in the graph if its node occurrence value is higher than a minimal configured value (*node_occ_th*) in the pruning algorithm.

To determine the elapsed time since a node was accessed, a new variable is associated with each node (*last access* field). When a node is accessed, this field gets the value of the global *access counter* mentioned above (which also increases its value). Thus, if the *last access* value of a node is lower than a configured access threshold (*access_th*) set in the pruning algorithm, then the node is pruned. Otherwise the node *last access* field is set to zero. When the

node pruning process finishes, the global *access counter* is reset, thus becoming ready to start a new learning process. When a node is pruned, all the arcs leaving that node or having this node as their destination are also removed.

Finally, notice that node pruning requires to walk on the entire graph because the nodes most likely to be pruned are the less requested ones.

Figure 9: Algorithm for pruning the RG graph

Input:
  rg: RG graph
  $referrer$: Referrer provided in the request
  th: Primary threshold
  $accesscounter$: Global counter of accesses
Output:
  rg: RG graph with improved knowledge
{ **PRUNING ARCS AND REDUCING OCCURRENCES** }
$a \leftarrow$ find node with $referrer \in rg$
if a occurrence $\geq node\_occ\_th$ **then**
  for all arcs $\vec{ar} \in a$ in rg **do**
    if $\vec{ar}$ transition confidence $< th$ **then**
      prune arc $\vec{ar}$
    else
      $\vec{ar}$ occ $\leftarrow \vec{ar}$ occ $* occ\_reduction\_factor$
    end if
  end for
  a occ $\leftarrow a$ occ $* occ\_reduction\_factor$
end if
Store a in rg
{ **PRUNING NODES** }
if $accesscounter \geq pruning\_th$ **then**
  for all nodes $n \in rg$ **do**
    if n occ $< node\_occ\_th$ or n last access $< access\_th$ **then**
      prune node n
    else
      n last access $\leftarrow 0$
    end if
  end for
  $accesscounter \leftarrow 0$
end if
return rg

6.3 Example of RG Pruning

To illustrate the pruning method, this section presents an example of a graph built by RG and the graph that results from the pruning of arcs and nodes. The graph built corresponds to the navigations performed by a set of clients to the Calendar website proposed in Figure 1.

In this scenario, navigation sessions have also been assumed to build the RG graph shown in figure 10(a). As known, nodes represent the visited pages, and arcs show the transitions among them. Each node shows the value of its occurrence and its *last access*, and each arc shows its occurrence and its transition confidence.

The corresponding navigations were as follows: many clients accessed the website by first visiting its main page, which is, consequently, very popular (see occurrence of *Calendars*). Few visitors were interested in the 2008 calendar or in the Mayan one. In fact, the last accesses to those pages were performed a long time ago (compare the global *access counter* with the corresponding *last access* value). The page showing the *Gregorian* calendar is quite popular but few users requested it before accessing the main page, possibly because one of the visitors shared the page URL with his or her friends, so the new visitors requested the *Gregorian* page directly, without requesting *Calendars* before. This can be observed in the graph by comparing the high value of the *Gregorian* node occurrence to the low occurrence value of the arc that links the *Calendars* node to the *Gregorian* one. The numerical values of these fields show possible consistent values with the users behavior described above.

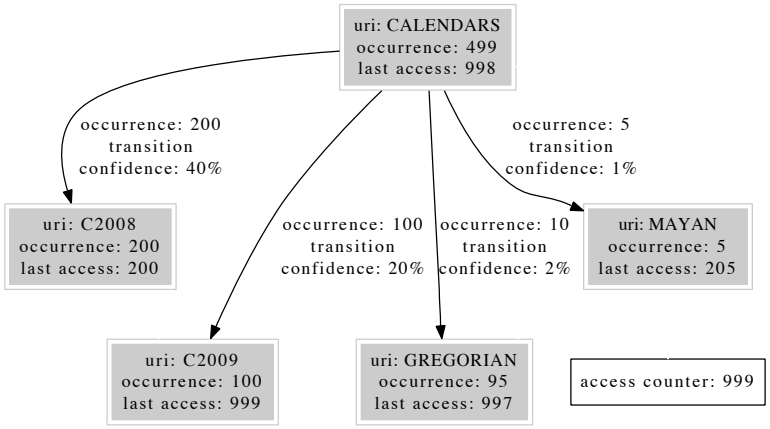
The pruning algorithm parameters for this demonstration are set as follows: $threshold = 15\%$, $node\_occ\_th = 50$, $occ\_reduction\_factor = 0.1$, $pruning\_th = 1000$, $access\_th = 500$. In this example, let's suppose that the RG algorithm receives a new prediction request for *Calendars*. Then, a prediction is performed, the graph is updated to reflect the new access, and the pruning mechanism is triggered.

The pruning process starts by checking the arcs that leave the node Calendar. Two of the arcs are pruned because their transition confidence is lower than the primary threshold: the arc to the *Gregorian* node, and the one to the *Mayan* one. Then, the remaining arcs and nodes update their occurrences using the reduction factor mentioned above. Then, when the global *access counter* reaches the pruning threshold (*pruning_th*), the process starts and all nodes in the graph are checked for pruning. Besides its popularity, the node *C2008* is removed because its last access was performed a long time ago. The *Mayan* node is also deleted but, unlike the previous one, it is deleted because it has a very low occurrence (which means it has not been popular since the last node pruning process). Of course, the arcs that arrive at or leave the pruned nodes are also removed from the graph. The graph resulting from the pruning is shown in Figure 10(b).

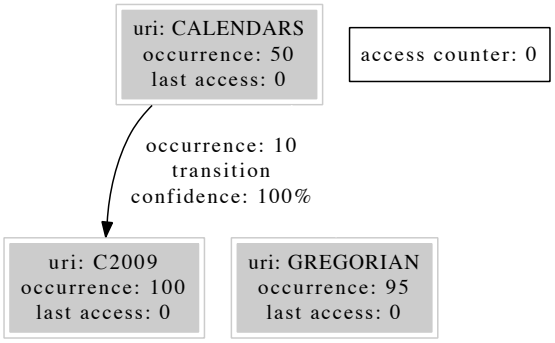
In summary, the initial graph of five nodes is reduced to only three nodes, and only one arc remains from the four initial arcs. Those elements of the graph were removed because they were not popular or had not been accessed for a long time.

6.4 Experimental Results of Pruning

This section evaluates the benefits of pruning. Graph complexity, prediction time, and page latency savings have been measured and compared to the original



(a) Before pruning



(b) Graph after arc and node pruning

Figure 10: Graph learnt and graph after pruning

Table 4: Distribution of the prediction service time among subtasks

Trace	**Algorithm**	**Data access**	**Learn**	**Prune**	**Predict**	**Prediction service time**
A	RG	0.630	0.739	—	0.303	1.672
A	RG+pruning	0.341	0.400	0.290	0.164	1.196
B	RG	0.095	0.112	—	0.046	0.253
B	RG+pruning	0.063	0.072	0.052	0.029	0.216

algorithm. To carry out the experiments, the *node_occ_th* and the *access_th* were set to 10 and 250 000 accesses, respectively. For the sake of clarity, the results of each trace are presented in a separate plot across the performed experiments because both traces present quite different characteristics.

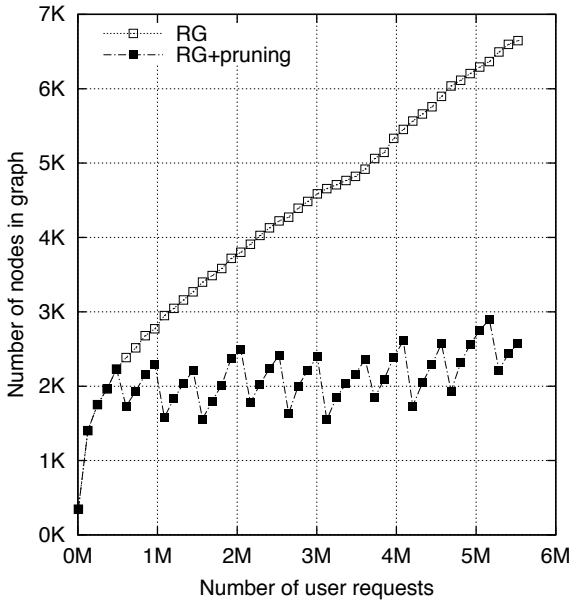
Figure 11 shows, for both traces, the increase in the number of nodes in the graph as the experiment progresses. Regardless of the trace, when using the original RG algorithm (upper curve), the number of nodes increases noticeably during the experiment. As expected, when applying pruning, the graph complexity is reduced each time pruning is performed.

Figure 12 illustrates how the number of arcs evolves in the graph when applying pruning and in the original algorithm. When pruning arcs, these are continually removed over the experiment, whereas when pruning nodes, these are removed at fixed intervals. The reduction in the number of arcs is caused by the periodic node pruning, because the removal also affects the number of arcs, that is, when a node is removed from the graph all its outgoing arcs are also removed.

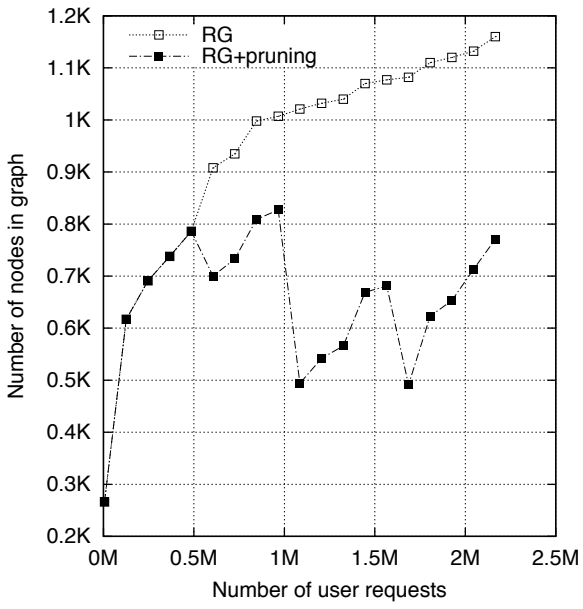
Figure 13 shows the average service time taken by the RG prediction algorithm to perform predictions. In addition to the learning and predicting phases, this time includes the pruning phase. In spite of this fact, compared to the original algorithm, the inclusion of the pruning mechanism significantly reduces the service time, which drops by about 30% in trace A and 23% in trace B (quantified at the end of the experiment).

These results mean that graph pruning does not have any adverse effect on the service time; on the contrary, it contributes to noticeably reduce it, thanks to the reduction of the graph complexity.

As mentioned above, the algorithm has five main subtasks. While four of them (access to data storage, learn from the users request, prune arcs and occurrences and, perform a prediction) run sequentially in the same process, the subtask of pruning nodes runs concurrently with the others and occurs periodically . Table 4 shows the weight of each subtask over the total prediction service time measured at the end of the experiments shown in Figure 13. As can be seen, data access and learning are the tasks that take more time, while predicting is much less time costly. When pruning is enabled, the service time increases but indirectly reduces the complexity of the other subtasks, and thus, decreasing the total prediction service time. This observation can be appreciated in both traces.

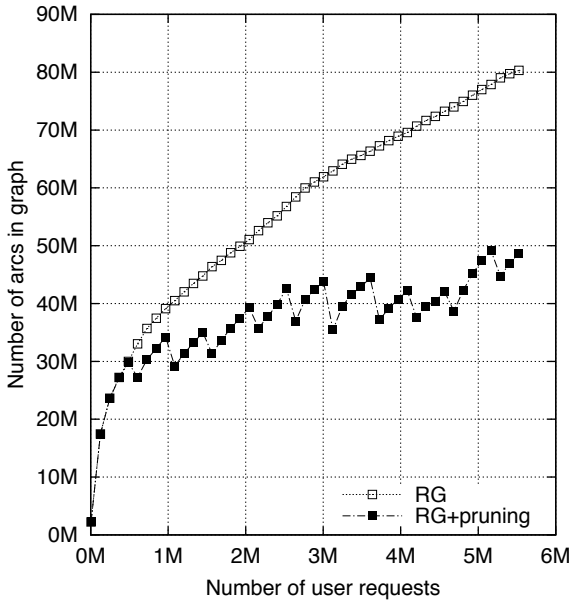


(a) Trace A

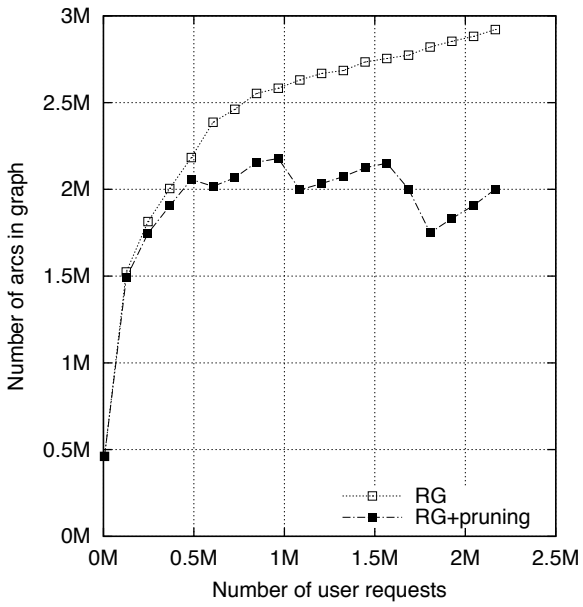


(b) Trace B

Figure 11: Number of nodes in graph when threshold is 0.1

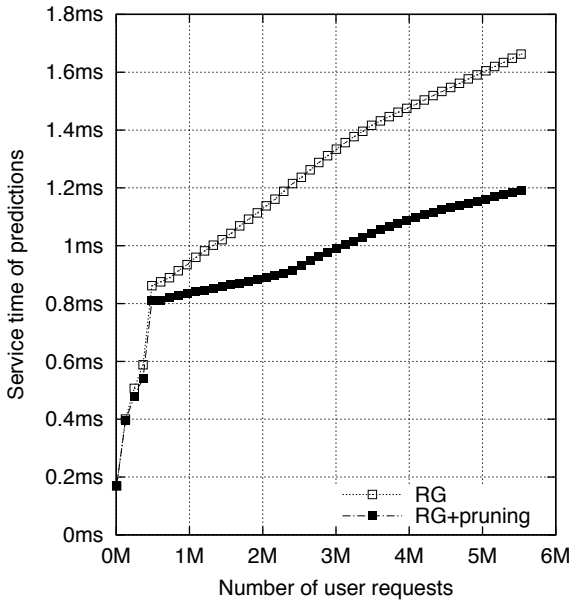


(a) Trace A

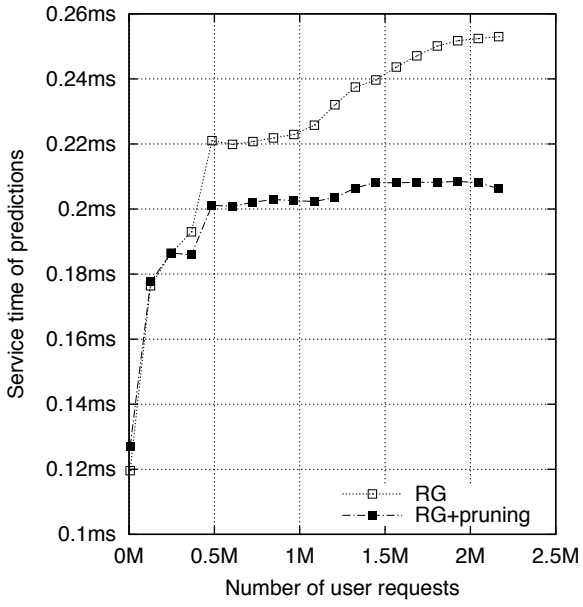


(b) Trace B

Figure 12: Number of arcs in graph when threshold is 0.1

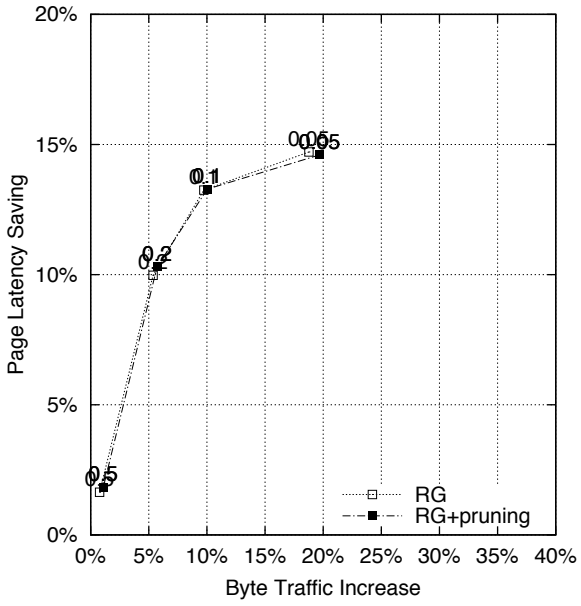


(a) Trace A

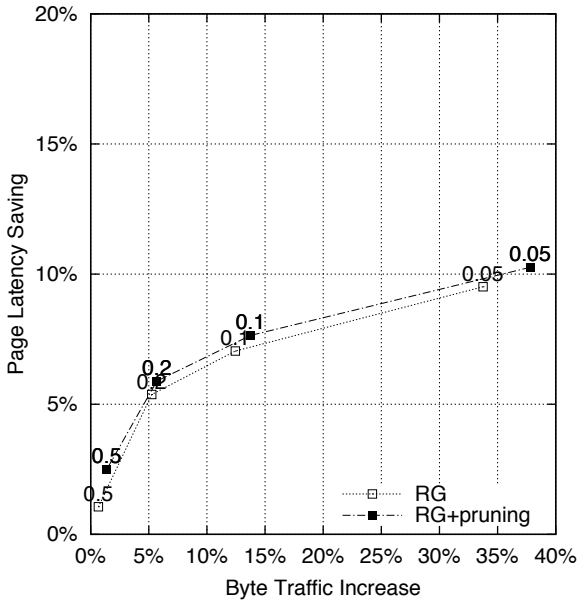


(b) Trace B

Figure 13: Prediction latency when threshold is 0.1



(a) Trace A



(b) Trace B

Figure 14: Page latency saving versus byte traffic increase

Finally, Figure 14 shows the page latency saving and byte traffic increase obtained by the RG algorithm with and without pruning. In trace A, pruning the graph results in almost no change in the performance achieved. It is observed that in trace B, RG obtains slightly higher benefit when pruning, but at expenses of increasing the cost. This is because pruning the graph deletes nodes and arcs with low transition confidence, and this fact increases the transition confidence of the remaining arcs. As a consequence, more hints are provided in the predictions, more hints are prefetched, and some of them finally result in prefetching hits.

In summary, the proposed method for pruning the RG graph allows us to reduce computational (i.e., less service time) and storage resources (i.e., graph complexity) without decreasing the original performance.

7 Conclusions

This paper has presented the Referrer Graph (RG) prediction algorithm and an associated prune mechanism, which is aimed to be a precise and simple solution for web prediction while consuming few computational resources. RG learns from user accesses and builds a Markov model, differentiating dependences on objects of the same page from objects of different pages. However, instead of using the sequence of user accesses as other algorithms do, RG uses the object URI and referrer associated to each request. This permits the design of a very simple algorithm because it does not need to keep track of previous user accesses (the user browsing session). It also means that RG establishes arcs to represent proven relations, instead of establishing arcs between objects for the circumstantial reason that they were requested sequentially by the same user. Consequently, RG establishes fewer arcs than other proposals, so the Markov model is smaller. This allows a faster management of the model when learning or predicting.

The experimental results show that RG compared to the best prediction algorithms proposed in the open literature, obtains a page latency saving similar, or even better when the traffic increase is a strong constraint, but requiring less computational and memory resources, thanks to its data structure and algorithmic simplicity.

In addition to the learning and prediction processes, a simple mechanism for pruning nodes, arcs, and occurrences in the graph has been devised for RG. This allows RG to learn from new user accesses over time without increasing the resource consumption. Results show that the proposed pruning mechanism significantly reduces the graph complexity and consequently the service time to perform a prediction, and has no adverse impact on the latency savings achieved, even improves them.

8 Acknowledgments

This work has been partially supported by Spanish Government Grant TIN2009-08201. The authors would also like to thank the technical staff of the School of Computer Science at the Polytechnic University of Valencia for providing us recent and customized trace files logged by their web server.

References

[1] V. Padmanabhan, J. C. Mogul, Using predictive prefetching to improve World Wide Web latency, Proc. of the ACM SIGCOMM Conference.

[2] T. Palpanas, A. Mendelzon, Web prefetching using partial match prediction, Proc. of the 4th International Web Caching Workshop, San Diego, USA.

[3] J. Domènech, J. Sahuquillo, J. A. Gil, A. Pont, The impact of the web prefetching architecture on the limits of reducing user's perceived latency, In J. Domènech PhD dissertation.

[4] J. Domènech, J. A. Gil, J. Sahuquillo, A. Pont, DDG: Efficient prefetching algorithm for current web generation, 1st IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), Boston.

[5] B. de la Ossa, J. Sahuquillo, A. Pont, J. A. Gil, An empirical study on maximum latency saving in web prefetching, Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'09) (2009) 556–559.

[6] E. Markatos, C. Chronaki, A top-10 approach to prefetching on the web, Proc. of INET, Geneva, Switzerland.

[7] A. Bestavros, Using speculation to reduce server load and service time on the www, Proc. of the 4th ACM International Conference on Information and Knowledge Management, Baltimore, USA.

[8] I. Zukerman, D. W. Albrecht, A. E. Nicholson, Predicting users' requests on the WWW, Proc. of the seventh international conference on User modeling (1999) 275–284.

[9] B. de la Ossa, A. Pont, J. Sahuquillo, J. A. Gil, Referrer graph: a low-cost web prediction algorithm, Proc. of the 25th Symposium On Applied Computing (2010) 831–838.

[10] J. Zhu, J. Hong, J. G. Hughes, Using markov chains for link prediction in adaptive web sites, in: ACM SIGWEB Hypertext, Springer, 2002, pp. 60–73.

[11] Q. Yang, J. Z. Huang, M. Ng, A data cube model for prediction-based web prefetching, Journal of Intelligent Information Systems 20 (1) (2003) 11–30. doi:http://dx.doi.org/10.1023/A:1020990805004.

[12] S. Gündüz, M. T. Özsu, A web page prediction model based on click-stream tree representation of user behavior, Proc. of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining.

[13] A. Nanopoulos, D. Katsaros, Y. Manolopoulos, A data mining algorithm for generalized web prefetching., IEEE Trans. Knowl. Data Eng. 15 (5) (2003) 1155–1169.

[14] D. Duchamp, Prefetching hyperlinks, Proc. of the 2nd USENIX Symposium on Internet Technologies and Systems.

[15] T. Ibrahim, C. Xu, Neural nets based predictive pre-fetching to tolerate www latency, Proc. of the 20th IEEE International Conference on Distributed Computing Systems, Taipei, Taiwan.

[16] B. D. Davison, Predicting web actions from html content, Proc. of the 13th ACM Conference on Hypertext and Hypermedia, College Park, USA.

[17] M. Deshpande, G. Karypis, Selective markov models for predicting web page accesses, ACM Trans. Internet Technol. 4 (2) (2004) 163–184. doi:http://doi.acm.org/10.1145/990301.990304.

[18] J. Domènech, B. de la Ossa, J. Sahuquillo, J. A. Gil, A. Pont, A taxonomy of web prediction algorithms, Expert Systems with Applications (0) (2012) –. doi:10.1016/j.eswa.2012.01.140.
URL http://www.sciencedirect.com/science/article/pii/S0957417412001583

[19] B. de la Ossa, J. A. Gil, J. Sahuquillo, A. Pont, Delfos: the oracle to predict next web user's accesses, Proc. of IEEE 21st International Conference on Advanced Information Networking and Applications, Canada.