Bayesian Network for Prefetching Caches

Alexandre de Spindler

Winter 2004

Contents

1 Introduction

There are various reasons in favour of caching and predicting the usage of information delivered on request. In-spite of continuously growing bandwidth and shrinking latency due to technological progress, the connection enabling access to an information system and request processing within the system still is a bottleneck. Moreover, since an information system is mostly shared by multiple users, its global availability benefits from reduced access.

Prefetching and caching are obvious choices to reduce information delivery latency caused by limited bandwidth and system resources. The information requested in the immediate future can be prefetched on connection idle time while the user is processing the previous delivery. The information likely to be requested again can be cached in order to avoid unnecessary requests to the system.

Caching also supports offline browsing which may reduce online time and bypass intended or accidental disconnection.

Additionally, prefetching can be used to suggest information of interest to a user if the prefetched items are likely to be requested next and hence reflect the users interests. It can help to gain knowledge about the information stored in the system, knowledge in form of link structures previously unknown but which have come out of usage. Thus, this knowledge can be used for the evolution of the underlying schema.

The user of an information system queries for objects containing the desired information. Those objects can be of arbitrary information granularity, e.g. they can contain a single phone number or they can aggregate a phone number to a name and a birth date, representing a person. When pursuing a particular interest, the user queries a subset of all objects specific to that interest. We refer to such a subset as a cluster. A user usually pursues multiple interests each one of them defining a cluster of objects. In traditional information systems, these clusters are unknown to the system. The user has to explicitly query for each single object of interest. We want to develop a system that recognises the user's interest (i.e. recognise a cluster given a member object) and that is able to find objects belonging to a given interest (i.e. find all member objects of a cluster). Additionally, the system should be sensitive to possible access sequences favoured by the user in case they arise.

In this work we present a prefetcher whose predictions are based on a probability distribution function (PDF) modeling the access to objects stored in an information system. The PDF is empirically approximated at the same time that the information is requested. Hence, a cache using our prefetcher not only stores the information objects most likely to be used again but also prefetches the objects most likely to be requested given the objects accessed before. Since the cache is bound to a given size, we are also interested in the objects with the least probability to be accessed in order to erase them from the cache. Hence, the prefetcher also names the objects that will be accessed with the least probability given the previous accesses. Our approach is based on the assumption that the user's access statistics reflect his/her interest (e.g. if a user accesses an object, he/she is

interested in the contained information)

We use the HTTP proxy proposed in [17] in order to test our prefetcher. The proxy is implemented such that prefetchers can easily be exchanged and hence allow quick evaluation and comparisons of our implemented prefetcher.

We will review related work in section 2 before presenting our approach in section 3. The implementation is sketched in section 4. We present and discuss our results in section 5 and propose future work in section 6. Appendix 7 gives a short introduction to Bayesian probability theory (as used throughout our work). In appendix 8 we give a quick and general introduction to Bayesian networks followed by the API of our cache system in Appendix 9.

2 Related Work

Since we are using belief networks to build a prefetcher, we need to consider two research areas as related work. Firstly, we investigate applications of belief networks within information systems and, secondly, we give an overview of current techniques of predictive prefetching and caching.

2.1 Belief Networks

Belief Networks have primarily been developed to model interacting and arbitrarily linked random variables. Given the observed values of a subset of the variables (evidence), the values of the remaining variables can be predicted based on their probability given the evidence. The network models the joint probability distribution (JPDF) over all variables involved (see appendix 7 for an introduction to Bayesian probability theory). Due to their probabilistic nature, belief networks handle uncertainty and noisiness in data well.

The two main representatives of belief networks are Markov random fields (MRF) and Bayesian networks. MRF are undirected graphs where vertices represent random variables and the edges dependencies. The fact that one variable is independent from all others except it's neighbours given these neighbours (a.k.a the Markov assumption of independence) facilitates the computation of the JPDF.

Bayesian networks were first proposed in [15]. [2] offers an overviewing introduction. We give an intuitive and general introduction to Bayes networks in appendix 8. As opposed to MRF, Bayesian networks are directed acyclic graphs where the vertices represent the random variables and a directed edge pointing from vertex a to vertex b can be read as "a causes b". As in MRF, the independencies modeled by the graph facilitate the JPDF computation.

The first famous application of Bayesian networks is QMR-DT, a decision theoretic reformulation of the Quick Medical Reference (QMR) model [16]. This network only consists of two layers, one representing (unobservable) deceases and the other representing (observable) symptoms. The causal connections go from the deceases to their respective symptoms (i.e. a decease causes it's symptoms). Given a set of observed symptoms (evidence), the network identifies the most probable decease. Both the structure and the parameters are static and were previously developed by experts.

The probably most widespread application of Bayesian networks is the office assistant in MS Office 97 and over 30 technical support troubleshooters (The Lumiere Project [9]). The static structure of the network contains variables representing system events, user behaviour and intentions as well as their causal interconnection. Evidence arises as the user is using the software which allows to predict the user's intention. Acquiring knowledge about the user's intention is an obvious requirement for offering appropriate support.

[11] makes use of Bayesian networks in order to acquire knowledge about the online audience of a particular website. In one study the structure was static and only the parameters were learned. The network contains (unobservable) variables

representing the user's satisfaction, the willingness to return, the willingness to recommend the website and the image the user has of the website owning company. The observable variables are the duration of the stay in each of the website's sections. The causal connections go from each unobservable variable to all observable variables. The parameters were learned using log files and represent the behaviour of 250 users. Once the parameters were learned the network could, for example, predict the user's satisfaction given the duration of stay in particular sections. In a second study, both the parameters and the causal connections were learned. The variables similar to the first study were previously given. In both studies, the network is used to relate predefined user variables (e.g. satisfaction, willingness to return) to observable user behaviour and to allow conclusions based on observed behaviour.

Markov random fields are widely used in segmentation and restoration of imagery [10]. Each pixel is represented by a random variable whose distribution function models the value of a predefined feature (e.g. luminance value, color distribution). The graph contains two vertices for each pixel: one representing the observed value and the other representing the unobservable true value (e.g. in a distorted image, the true value of a pixel feature is it's value in the original undistorted image). The assumption is that the value of a pixel is independent from all other pixels given it's neighbours. The neighbourhood of a pixel is represented in the graph as the set of adjacent vertices. Both the connections and the vertices are learned from data, and the JPDF can be used either to reconstruct and classify an image or to recognise a feature in the image.

MRFs (as well as Bayesian networks) have also been used to implement probabilistic relational models (PRM) [7]. A PRM defines a probability distribution over a database describing the relational schema of the domain and the probabilistic dependencies between the attributes. As opposed to traditional relational models, where a type has a set of attributes and an instance of that type has a value assigned to each attribute, in PRM attributes are assigned to a type according to a probability distribution as well as the value of an attribute of a particular instance.

The belief network contains vertices representing attributes and relations. The edges represent dependencies. The structure and the parameters are learned from the data by querying the database and using the approximation equality 12 presented in appendix 7. Hence, the schema can be computed given the data and arising relations are used for link prediction.

PRMs have also been used for hypertext classification [8]. In this case, the possible entities in the relational model were restricted to be either of type *page* or *link*. The type *link* contains two attributes, one referring to the page containing the link and the other referring to the page the link points to. The link is modeled to depend on the content of both pages which is stored as a member bag of words in the type *page*. Again, the parameters are learned from data, i.e. by counting the number of times a page links to another and by using this value to approximate the CPDF. Once the parameters have been learned, the probability of a link given a set of observed words can be computed. The type *page* also contains a member

label which is used for classification.

2.2 Predictive Prefetching and Caching

[1] offers an overview covering current prefetching techniques. Prefetching is closely related to a wide variety of research areas such as web, database, network, data mining and algorithmic. [19] proposes a classification scheme for web mining techniques which can as well be used for prefetching techniques.

On one hand, prefetching varies depending on the kind of information that is (pre)processed. Techniques adopted from information retrieval go as far as parsing the content of an object in order to find similar objects. Other techniques make use of structural information (e.g. link structures, relations) between the objects. Finally, the usage statistics are analysed in prefetching techniques drawn from machine learning.

On the other hand, prefetching varies according to the technique used to discover and analyse patterns. These include automatic schema evolution, clustering, classification, sequence pattern recognition and Bayesian statistics (i.e. CPDFs). Finally, the prediction of the object to be prefetched can be based on a single user or averaged over multiple users.

[4] compares different data compression techniques that have been used for prefetching. Effective data compression is achieved by encoding with few bits data expected with high probability and with many bits data expected rarely. The compression engine builds up a dynamic probability distribution for the data to be compressed. For prefetching, the sequence of accessed objects is treated as a sequence of bits to be compressed. The resulting distribution is used to determine the object(s) with high expectancy.

[3] proposes to predict a user's intention using naive Bayesian classifier. In a first step features are extracted from the objects using information retrieval techniques. Each object is labeled with the respective user intention. The second step consists in learning the CPDF $P(Intention|Feature)$ by counting. Now that the parameters have been learned, the CPDF can be used to predict the most probable intention given a set of observed features (e.g. features of the previously requested object). For prefetching, the intention to be predicted is the object requested next and thus the learned CPDF translates into $P(NextRequest|FeaturesOfPreviouslyRequestedObject)$

Instead of extracting features, [17] computes the CPDF $P(PredictedRequest|PreviousRequest)$. Hence, each request r is treated as evidence to select the respective CPDF $P(PredictedRequest|PreviousRequest = r)$ which defines a probability for each object in the domain of $PredictedRequest$. In both approaches, the CPDFs can be approximated using batch or online learning.

The CPDF $P(PredictedRequest|PreviousRequest)$ can be enhanced to be conditioned on the last k requests instead of the last request only. [5] compares various prefetching methods based on the Markov assumption of independence. A Markov tree of depth k is proposed to comprise all CPDF based methods.

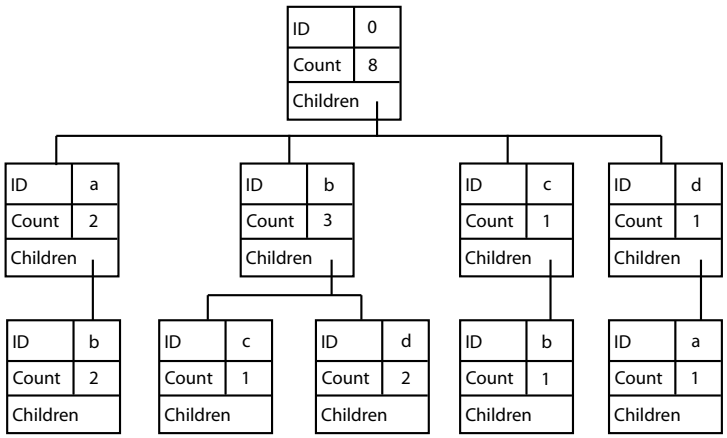

Figure 1: Markov Tree with Depth 2 after the Sequence of Requests a,b,c,b,d,a,b,d

The main idea is that each node in the Markov tree represents a request and counts the number of times it has been called by the user (n_{node}). The root node represents the empty request. It's children are the first requests in any sequence of requests that have occurred. For all first requests, all requests that have appeared as a successor are added to the set of their children. If we stop here, the resulting Markov tree is said to be of depth $k = 2$. For the third request in a sequence we add a node $request_2$ representing it's predecessor (the second request) to the children of the root node and add a new node $request_3$ to the children of $request_2$. We proceed accordingly for every request in the sequence: For any pair of requests following each other in a sequence we either create the new (parent,child) pair in case this sequence has not appeared yet or we simply increment the corresponding counters if the nodes are already present. If the parent of a pair is already contained in the children set of the root node but the succeeding request has never been it's successor before, we add the new successor to the children set of the parent request. Figure 1 shows a Markov tree with depth $k = 2$ that arose from the sequence of requests a, b, c, b, d, a, b, d. The pairs contained in that sequence are $(a, b), (b, c), (c, b), (b, d), (d, a), (a, b), (b, d)$.

Using the the number of times the nodes have been called we can easily compute the CPDF $P(Child|Parent)$ as $\frac{n_{child}}{n_{parent}}$ for all children of the node contained in the children set of the root node representing the current request. [17] then selects the child with the maximum CPDF value. Our example sequence would yield $P(b|a) = \frac{2}{2} = 1$ and $P(c|b) = \frac{1}{3} = 0.33$.

We can increase the depth k to have a probability distribution for each request conditioned on it's k predecessors. Figure 2 shows the Markov tree that has been build after the same sequence of requests we used before. Now we can compute $P(c|a, b) = \frac{1}{2} = 0.5$ or $P(b|b, c) = \frac{1}{1} = 1$.

A Markov tree can be seen as a Bayesian network where the structure, as well as the CPDFs, are learned online. In this sense our approach is almost similar to the general Markov tree with the difference that we use belief propagation in order to compute the CPDF $P(PredictedRequest|kPreviousRequests)$ for every requestable object instead of only those objects for which a CPDF conditioned on

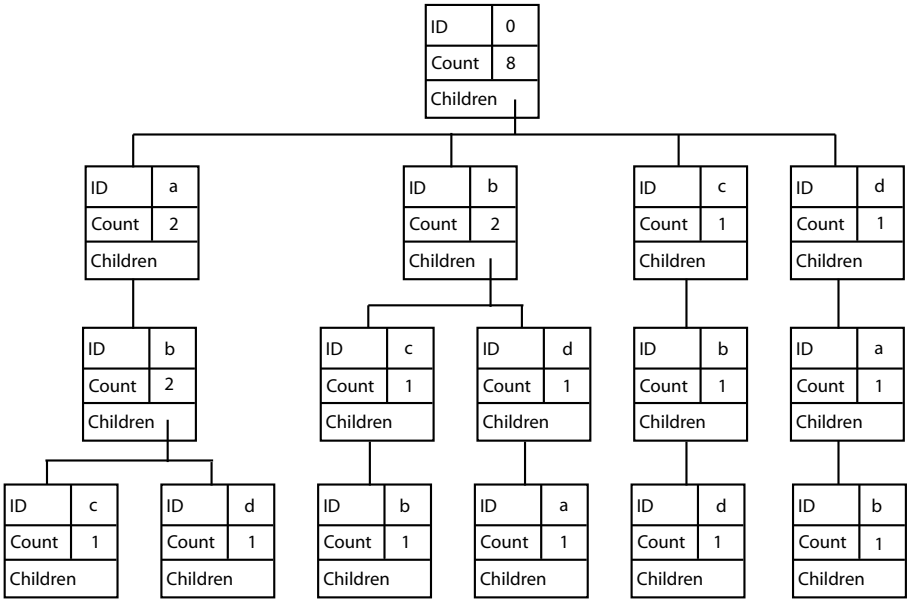

Figure 2: Markov Tree with Depth 3 after the Sequence of Requests a,b,c,b,d,a,b,d

the currently last k requested objects exist. For example, let us assume a sequence where the last two requests were a and b. Using the Markov tree in figure 2 for predicting the third request we only have the CPDFs $P(c|a,b)$ and $P(d|a,b)$. Thus, we can only predict the likelihood of the two objects b and c. In a Bayesian network we could compute the CPDF values conditioned on (a,b) for all objects in the network. In the Markov tree we only have the CPDFs conditioned on (a,b), (b,c), (b,d), (c,b) and (d,a). In a Bayesian network we can condition a probability on an arbitrary set of objects without explicitly modeling every possible sequence of requests.

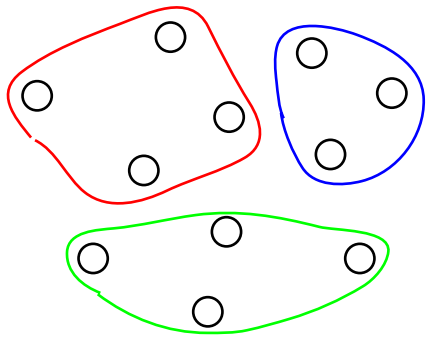

Figure 3: Objects Stored in an Information System

3 Our Approach

Our approach is based on CPDFs that express the likelihood of the access to an object given the history of previous accesses. The CPDFs are computed using belief propagation in a Bayesian network. The structure and the parameters of the network are learned in an online manner.

The main goal of our prefetcher is to recognise groups of objects (clusters) that are requested simultaneously when a user pursues a particular interest. When the user requests an object of an arbitrary cluster, other member objects of the same cluster will be prefetched. Therefore, once the cluster structure is known, we first need to be able to recognise which cluster has an arbitrary object assigned to it and, secondly, we need to be able to retrieve all member objects of an arbitrary cluster. We assume that the user requests reflect his/her interests, i.e. if a user requests a particular object, he/she is interested in it's contained information. The interface definition of a prefetcher restricts the available information about the user, his/her interests and the objects stored in the information system to access statistics: Each time the user requests an object, the prefetcher is told the object identity of the currently requested object. Given a current request, the prefetcher is expected to name at least one id referring to an object that

1. has been requested at least once in the past (i.e. the prefetcher knows about the object)

2. has the highest probability of being queried by the user in his/her next request

In other words, the access probability of an object should reflect the cluster structures. An object which is a member of the same cluster as the currently accessed object should have a higher access expectation than an object from a different cluster. If the currently accessed cluster contains k objects, then the $k-1$ most probable objects should amount to all members of this cluster except for the currently accessed object.

In figure 3 example objects stored in an information system are represented as black circles. The coloured borders denote the clusters that are unknown to the information system and which the prefetcher should learn to recognise.

In this section we first intuitively describe the general idea of our approach followed by a detailed description of how we use a Bayes network to perform prefetching.

3.1 General Idea

For any sequence of requests we define o_t as the currently accessed object, o_{t+1} as the next requested object and $o_{0..t}$ as the set of all objects that have been accessed so far including the current request for o_t. O is the set of all objects contained in the information system. Hence, $O - o_{0..t}$ is the set of objects that have not been accessed so far. We use the superscript notation o^i to refer to the object i. If the currently accessed object has the id i then $o_t = o^i$.

We use the notation $state(o_i) = 1$ to express that the object o_i has been accessed at least once before (and vice versa for $state(o_i) = 0$). In other words: $\forall o^i : o^i \in o_{0..t} \implies state(o^i) = 1$ and $\forall o^i : state(o^i) = 0 \implies o^i \in O - o_{0..t}$.

Finally, we denote $n_{condition}$ as a counter that counts the number of times $condition = true$ has been observed.

For each object o^i we have a CPDF $P(state(o^i) = 1|o_{0..t}, O - o_{0..t})$. To keep notations short, we will write $P(state(o^i) = 1|o_{0..t})$ and any condition on the states of the objects in $o_{0..t}$ implicitly implies the inverse state on the objects in $O - o_{0..t}$. $P(state(o^i) = 1|o_{0..t})$ is the probability that the object o^i will be queried in the next request given all previously accessed and unaccessed objects. In order to fulfill the requirements stipulated above, this probability should be high if o^i and the objects in $o_{0..t}$ are members of the same cluster and it should be low otherwise. We explain how we approximate this CPDF empirically such that it reflects the clusters.

In general, a user interacts with the information system by sequentially requesting objects. In this section we assume that the sequence of requests is finite and that we know when a sequence ends. We call such a finite sequence of requests a session. Multiple sessions may follow each other but each session can clearly be delimited. In the next subsection we derive the computation of $P(state(o^i) = 1|o_{0..t})$ assuming that the user pursues one interest per session, i.e. each object requested within a session belongs to the same cluster. In the subsection after that we derive the computation in case a user pursues multiple interests within a session.

3.1.1 One Interest per Session

Figure 4 shows the situation after a session. We have added the ids of the objects written as a superscript next to the circle. The user has been interested in the topic covered by the cluster marked red in figure 3 and requested all members. At the end of the session we update the counters that are used to compute $P(state(o^i) =$

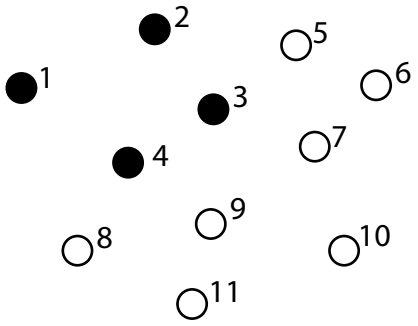

Figure 4: Objects accessed within one session. The user pursues only one interest per session. The objects in black have been accessed. The numbers denote the object ids.

$1|o_{0..t}, O - o_{0..t})$ for each object o^i. Using equation 12 defined in appendix 7, the CPDF of o_i can be written as

$$P(state(o^i)|o_{0..t}) \quad = \quad \frac{n_{state(o^i)=1 \ \& \ \forall \ o^j \in o_{0..t}:state(o^j)=1}}{n_{\forall \ o^j \in o_{0..t}:state(o^j)=1}} \tag{1}$$

$n_{state(o^i)=1 \ \& \ \forall \ o^j \in o_{0..t}:state(o^j)=1}$ is the number of times that o^i has been accessed while all previously accessed objects in $o_{0..t}$ were marked as accessed simultaneously. $n_{\forall \ o^j \in o_{0..t}:state(o^j)=1}$ is the number of times that all objects in $o_{0..t}$ were marked as accessed simultaneously. Each object has a counter for all possible combinations of states every other object may be in. In our example we have eleven objects each one of them able to be in two different states (accessed and not accessed, 0 and 1). Hence, every object has 2^{11-1} counters, each counting the number of times the combination of states particular to that counter has occurred.

For all objects in our example information system, the counter $n_{state(o^{1..4})=1 \ \& \ state(o^{5..11})=0}$ is increased. As a result of this increment, $P(state(o^i) = 1|state(o^{1..4}) = 1 \ \& \ state(o^{5..11} = 0)$ will be higher for all $o^i \in$ red cluster than for the members of other clusters. This is due to the fact that for any member o^i of the red cluster, $P(state(o^i) = 1|state(o^{1..4}) = 1)$ has increased while for the members o^j of all other clusters the increment of the counter increased $P(state(o^j) = 0|state(o^{1..4}) = 1)$. Since

$$P(state(o^j) = 1|state(o^{1..4}) \quad = \quad 1 - P(state(o^j) = 1|state(o^{1..4}) \tag{2}$$

the increment actually decreased $P(state(o^j) = 1|state(o^{1..4})$.

This means that anytime members of the red clusters are marked as accessed, the CPDF of any other member of the same cluster will be higher than the CPDF of the members of all other clusters. Figure 5 shows the access statistics at the end of a second session. At the end of the previous session, the states of all objects have been reset to zero. This time the user queried the objects of the blue cluster. Proceeding as before to update the CPDFs of all objects we can now

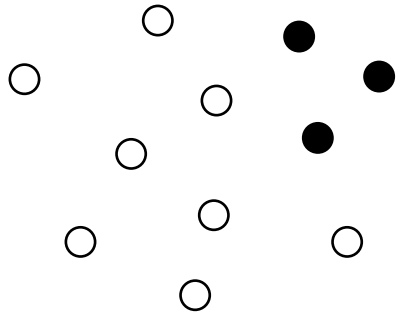

Figure 5: Objects accessed within one session. The user pursues one interest per session. The objects in black have been accessed.

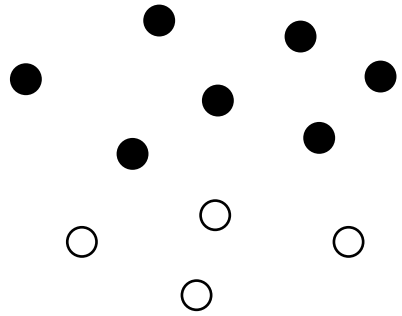

Figure 6: Objects accessed within one session. The user pursues multiple interests per session. The objects in black have been accessed

differentiate between all three clusters. Hence, the CPDFs successfully reflect the cluster structure induced by the user's interest.

3.1.2 Multiple Interests per Session

Figure 6 shows the states of the objects in our example information system after one session in which the user pursued two interests. As opposed to the previous example the objects of two clusters have been accessed.

Just like before, we compute the CPDF $P(state(o^i) = 1|o_{0..t})$ of all objects o^i by counting. This time, this will not only increase the CPDFs of the objects in the red cluster but also that of the members in the blue cluster. Therefore, these two clusters can be differentiated from the green cluster, but they are treated as one cluster. In this case we need more sessions.

Figure 7 shows the situation at the end of the session following the one in figure 6. Note that between the two sessions, the states of all objects have been reset to zero. Again we update the values of all CPDFs. This time, the CPDFs of the members of the red and green cluster are increased. We note that the CPDFs of the objects in the red cluster have been increased twice so far. Thus, the objects of the red cluster can now be differentiated from the ones in the blue cluster. If

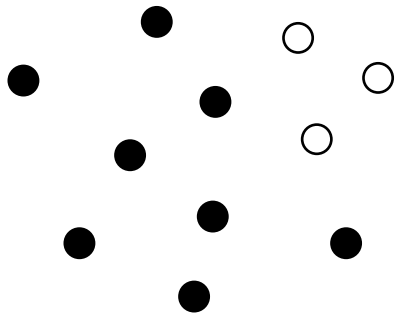

Figure 7: Objects accessed within one session. The user pursues multiple interests per session. The objects in black have been accessed

all of the user's interests are equally probable to be pursued within a session and independent from each other, then the CPDFs will eventually be able to recognise the clusters. In contrast to the single interest per session scenario, we need more sessions in order to make the CPDFs reflect the cluster structure. It is required that a sequence of requests induced by an interest of the user is always completely processed within one session. There may be multiple interests per session but each one of them must be fully processed at the end of a session when the CPDFs are updated.

3.1.3 Midterm Conclusion

In both cases (single/multiple interest(s) per session), the CPDFs reflect the cluster structure entailed by the users interests. If we want to recognise the user's current interest, we can at any time (i.e. before, within and after a session) multiply all current CPDF values of all member objects for each cluster by each other. The cluster with the highest probability value is the one containing the objects satisfying the user's current interest.

If we want to find all member objects of the cluster currently visited by the user, we take the k objects with the current highest probability given the previously accessed objects. We have seen before that the access to an object of an arbitrary cluster C increases the CPDFs $P(state(o^i) = 1|o_{0..t})$ of all $o^i \in C$ while it decreases the CPDFs of the members of all other clusters.

Using this update schema for all CPDFs in an information system spawns a prefetcher that fulfills our requirements stipulated in the beginning of this section. All we used is the id sequence of the accessed objects and the knowledge about the end of each session. We needed the session delimiters in order to know when to update the CPDFs and when to reset the states of the objects to zero. Obviously, if all counters are increased after each request and we never reset the states, we only gain information about which objects are never queried. All other objects will be treated as one cluster.

The need to delimit the sessions is an unacceptable requirement since it either requires heavy use of the information contained in the objects or additional user

interaction and thus extensions to the interface of our prefetcher. To avoid this dilemma, we propose some work-arrounds in section 3.2.5.

An additional detail that was implicitly imposed using the technique presented above is that the prefetcher needs to maintain 2^{n-1} counters for each of the n objects contained in an information system. Obviously this requires an unreasonable amount of storage space. This is why we decided to use Bayesian networks with which any CPDF can be computed within reasonable computational costs.

3.2 Bayesian Networks

The main task for a Bayesian network is to model a JPDF over all involved variables so that any CPDF can be computed.

We treat $state(o^i)$ of every object as a random variable whose value can be either observed or predicted. If an object o^i has been accessed, the respective variable is considered as evidential variable and the value $state(o^i) = 1$ is our evidence. The CPDFs $P(o^i|evidence) = P(o^i|o_{0..t})$ can be computed using belief propagation and will be used for predicting the next access.

The CPDFs required for belief propagation can be estimated by counting like we did in section 3.1. Building up the structure (i.e. defining the dependencies) requires some additional considerations. For belief propagation we used loopy belief propagation [12] which is based on Pearl's polytree algorithm [15].

In this section we present the details of using a Bayesian network for prefetching in terms of the problems we encountered and the solutions we propose. From now on we will use o^i to denote the random variable modeling $state(o^i)$ in the network. In the following two subsections we keep the assumption that we know when a session ends. We then explain how our prefetcher is able to renounce assuming explicit session delimiters in section 3.2.5.

3.2.1 Structural Learning

The structure of a Bayes network consists of variables and dependencies. The variables can be learned in a straight forward manner: every time an object is requested the network checks whether the respective variable is contained in the graph or not. If it is not then it will be added and if it is the structure remains unaffected.

In a series of n requests $o^1, o^2, .., o^n$ we say that o^i causes o^{i+1} for every $i < n$. Hence, we always connect the currently accessed object o_t to it's predecessor o_{t-1}. In case o_t has already been part of the network and o_{t-1} has been it's predecessor at least once before, then nothing is done. If the pair (o_t, o_{t-1}) has never appeared in the sequence then an edge is added pointing from o_t to o_{t-1}.

The structural learning process initiated by each access ensures that the accessed object is contained in the network and that it is connected to its predecessor. This strategy of structural learning will naturally lead to a graph structure representing the observed access sequences. Figure 8 shows an example network. For now we ignore the nodes and edges in grey. The black nodes and edges have been inserted to the network due to the sequence $o^1, o^2, .., o^n$.

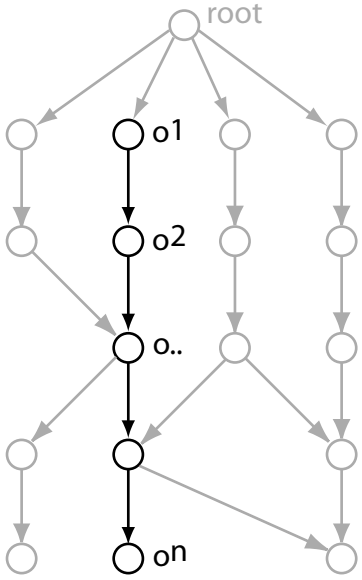

Figure 8: Example Structure of a Bayes Network

A Bayesian network is a directed acyclic graph. In order to achieve proper belief propagation, the graph must be connected. If we have two sequences that do not share any common request we must still connect them. We chose to have a root node in the network which is treated as an ancestor of every first request in any sequence. This root node does not represent any request.

We avoid creating a directed cycle whenever a variable o^i is to be made dependent on a variable o_j by checking if o^i is already an ancestor of o^j. If this is the case, the two are not connected.

3.2.2 Parameter Learning

In section 3.1 we have seen how a CPDF can be approximated empirically by means of counting. For belief propagation we need the CPDF $P(o^i|parents(o^i))$ for every variable in the network (the function $parents(o^i)$ returns the set of variables on which o^i is modeled to be dependent from by an edge). Usually the parents contain a lot less variables than all the variables in the network. This is a significant improvement to the technique introduced in section 3.1 which implied a complete CPDF conditioned on all other variables for each variable. For the Bayesian network every node maintains counters for every possible combination of states it's parents may be in. Every time a node is told to update it's CPDF it determines the states of it's parents and increases the respective counter. Whenever a new parent is added to a variable the number of counters have to be doubled.

If a variable o^i does not have any parents then the belief propagation only requires the prior probability distribution $P(o^i)$. Therefore, such a variable only maintains two counters counting the number of times the variable has been accessed and the number of times it has not been accessed.

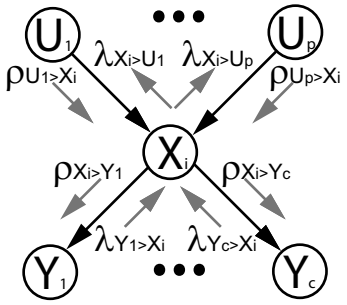

Figure 9: Messages computed, Sent and Received during Belief Propagation

During a session, one node is set to have been accessed after each request. At the end of a session all variables update their CPDF, i.e. they increase the counter representing the current state of itself and its parents (if there are parents) and the state of all variables is reset to zero.

3.2.3 Belief Propagation

In this section we give an overview of the formulae we use for belief propagation. We have used Pearl's polytree algorithm which was introduced in [15] to implement loopy belief propagation. In [18] we present Pearl's algorithm in its full details. Loopy belief propagation is an approximate inference algorithm that has been showed to work well for Bayesian networks with loops of arbitrary size if the prior probabilities are not too small (small $\sim$ of the order of 10^{-3}). Though, the algorithm may oscillate between values that are uncorrelated with the correct CPDFs [12].

$$p(x_i|e) \propto p(e_i^-|x_i)p(e_i^+, x_i) =: \lambda_i(x_i)\rho_i(x_i) \tag{3}$$

$$\rho_i^t(x_i) = \sum_u p(x_i|u \cup e_i^+) \prod_{j=1}^{p} \rho_{U_j \to X_i}^t(u_j) \tag{4}$$

$$\lambda_i^t(x_i) = \prod_{j=1}^{c} \lambda_{Y_i \to X_i}^t(x_i) \tag{5}$$

$$\rho_{X_i \to Y_j}^{t+1}(x_i) = \rho_i^t(x_i) \prod_{k \neq j} \lambda_{Y_k \to X_i}^t(x_i) \tag{6}$$

$$\lambda_{X_i \to X_i}^{t+1}(x_i) = \sum_{y_j} \lambda_{Y_j}^t(y_j) \sum_{v_1,\ldots,v_q} p(y_j|\pi_{Y_i}) \prod_{k=1}^{q} \rho_{V_k \to Y_j}^t(v_k) \tag{7}$$

The goal of belief propagation is to compute $p(o^i|o_{0..n})$ for every variable o^i. In the formulas above we use the notation x_i to denote the ith value of the random variable X and e_i is the ith value of an evidential variable. Thus $p(o^i|o_{0..n})$ is written as $p(X|E)$. X is the random variable for which $p(x_i|e)$ is currently computed, U is the set of its parents and Y is the set of its children (E is the set of

evidential variables, all o^i with $state(o^i) = 1$ in our case). Figure 9 depicts the family relations of a variable X.

Every node computes its λ and ρ values according to equations 4 and 5. Once those values are computed, the node can compute the λ and ρ messages it will send to its parents and children according to equations 6 and 7. The λ and ρ values are a function of the λ and ρ messages received from its children and parents. Figure 9 illustrates the messages computed and the nodes they are sent to. This circular interdependencies between the values and the messages would cause deadlock situations if the network is not a polytree and the nodes would wait for all messages before computing the values [15]. In loopy belief propagation the messages are initialised with vectors of 1s. Thus, all messages exist from the beginning and every value can be computed. The values and messages are computed iteratively, i.e. with every additional message received their values converge towards a fixpoint [12].

The algorithm stops as soon as no CPDF value $P(X|E)$ changes by a value greater than a margin given as a parameter.

3.2.4 Querying the Belief Network for Access Prediction

For predictive prefetching there is one very important query to the belief network: Which is the most probable object accessed next given the previous sequence of accesses. Since every previously requested variable has been set to *accessed* we can initiate a belief propagation in order to determine the CPDF $P(o^i|o_{0..t})$ for every variable o^i. We then go through all CPDFs and select the k variables with the highest probability given the evidence $o_{0..t}$.

In order to keep the size of the cache smaller than its capacity, the Bayes network is asked to return the k least probable objects. This functionality has not been implemented in this work.

There is more information contained in the network modeling the JPDF over all variables. Since we can compute every CPDF for any set of variables conditioned on any other set of variables, we could look for structure within the objects in the information system. One one hand, this structural information can be used to maintain the Bayesian network, in particular its structure. On the other hand we could also model the objects in the information system in terms of schema evolution. In section 6 we propose some possible usage of the structural information.

3.2.5 Special Issues on using a Bayesian Network for Prefetching

The main problem of using Bayesian networks to model a JPDF over variables representing requests to an information system is that we do not know when a user stops pursuing a particular interest and starts pursuing another. In section 3.1 we have shown that session delimiters are used to update the CPDFs, reset the evidence and to treat the first request of a succeeding session as if it did not have a predecessor.

The problem is that we do not know when a session starts or ends. Following we enumerate the work-arrounds we used to make up for this lack of knowledge.

- The CPDFs of every variable is updated after each request instead of at the (unknown) end of a session.

- The evidence decays after a number k of requests instead of being reset at the (unknown) end of a session. Hence the CPDFs $P(o^i|o_{0..t})$ resulting from the belief propagation are conditioned on k variables. This corresponds to a Markov tree with depth k as presented in section 2.

- The third issue is when to stop making a variable depend on its predecessor. Ideally, this would be at the (unknown) end of a session. One way would be to force a session ending after every lth request.

  Another possibility is to use randomness in order to account for the uncertainty about which object causes a current request. In section 3.2.1 we assumed that the predecessor within a sequence of requests to cause its successor. This does not account for the possibility that a user changes its pursued interest within a sequence or that a request may cause multiple successors in parallel, which still appear in a row in the sequence of requests. In order to include those possibilities the ancestor is chosen randomly from the d last requests plus the root node. The probabilities for the previous request, the root node and the other $d-2$ previous requests to be selected can be set individually. This technique leads to a connected graph with the nodes representing objects from the same cluster being strongly interconnected and nodes form different clusters being loosely connected. Since we use randomness it is possible that we do not capture the true cause of a request. In appendix 8 we stated that variables can exercise influence on each other although they are not directly connected by an edge. Thus, we assume that it is not vital to the correct representation of the JPDF in the network that we capture all dependencies using edges. Dependencies between variables arise even though they are not connected. In section 6 we propose to investigate on the importance of correctly representing dependencies using connections to the quality of the JPDF. We also propose to periodically analyse the network and/or add and erase edges. That way the structure of the network can be corrected to better capture the true dependencies.

  We try both approaches (the forced session ending after every lth request and the random ancestor selection) and discuss the results in section 5.

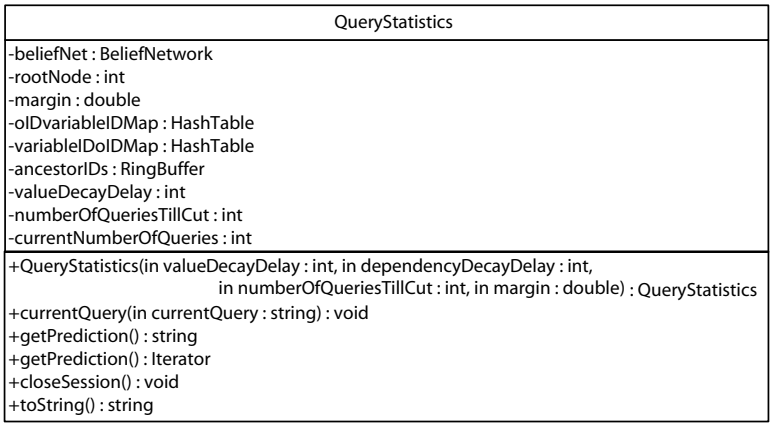

Figure 10: The Query Statistics Class. The set and get methods for the members are omitted.

4 Implementation

The prefetcher we presented in the previous section maintains access statistics in order to propose objects to be prefetched. In the architecture of the HTTP proxy server used in this work it does not prefetch objects itself. Our prefetcher only receives the id of the currently retrieved object and returns a list of ids to be prefetched. Further details about the architecture and workflows in the HTTP proxy server can be looked up in [17]. In this section we present the implementation of our prefetcher by describing the different classes and their interaction.

Throughout this section, in order to avoid name conflicts within the proxy server framework, our prefetcher will be referred to as query statistics.

4.1 Query Statistics for Prefetching

Figure 10 shows the UML diagram of the QueryStatistics class. This class maintains the access statistics and returns either the id of the object with the highest probability of being requested or a list of all known ids sorted according to their probability of being requested. The constructor takes the following parameters:

- *int valueDecay* is the number of requests the state of a variable is kept as evidence before it is reset to zero. The default value is 3. If this value is set to 0 the state decays and reactivates automatically and randomly. We use this behaviour for the root node in order to keep it as uninformative as possible.

- *int dependencyDecay* is the number of predecessors of a variable from which one is selected to be its cause. At every request, the current variable is modeled to depend on one of these predecessors, chosen at random. If this value is set to zero then the current variable is always connected to its immediate predecessor. If this value is set to one then the current variable is always connected to the root node. For all values greater than one, a parent is

chosen at random according to the following probability distribution: with a probability *pRootNode* and *pMostRecentAncestor* the parent is either the root node or the immediate predecessor respectively. The other predecessors might be chosen to be the parent with a probability of *pBetween*. In section 6 we propose a continuous distribution function to select the parent. These three probabilities can be set individually. An additional boolean parameter allows to specify whether the root node should always be kept as a potential ancestor or if it is chucked out at the *dependencyDecay*th request.

By default, dependencyDecay is set to 3.

- *int numberOfQueriesTillCut* is the number of requests during which every variable is made dependent from (one of) its ancestors. After every $numberOfQueriesTillCut$th request all evidence is reset and the next request is treated as if it did not have any predecessor (forced cut, entails the invocation of *closeSession()*). If this value is set to zero, then a cut will never be forced. Zero is the default value.

- *double margin* is the terminating condition for the loopy belief propagation. As soon as none of the CPDFs $P(X|E)$ changes by a value greater than $margin$ the propagation is stopped. This value is set to 0.001 by default.

All four of those parameters can be set using the graphical user interface of the proxy server. Besides the integer and double members to store the values received by the constructor, we have the following private members:

- *BeliefNetwork beliefNet* is the Bayesian network that models the JPDF over the states of all objects.

- *HashMap oIDvariableID, variableIDoID* manage the mapping from the object id to the variable id used in the Bayesian network. In our case, the object id is a string and the variable id is an integer number. Since the java library class HashMap only allows the query by key and not by value, we have to maintain two HashMap instances.

- *RingBuffer ancestorIDs* is a ring buffer that always contains the last *dependencyDecay* accessed variable ids and the root node or simply the last accessed variable id (depending on the value of the parameter *dependencyDecay*). This allows to make the currently accessed variable dependent from one of the *dependencyDecay* last variables or the root node, chosen at random. This ring buffer is emptied by the method *closeSession()*.

- *int currentNumberOfQueries* is a counter variable used to determine when $numberOfQueriesTillCut$ requests have been processed. If $numberOfQueriesTillCut > 0$ it is increased after each request. As soon as $currentNumberOfQueries > numberOfQueriesTillCut$, *closeSession()* is invoked. This value is set to 1 by the method *closeSession()*.

The interface methods used for prefetching are *void currentQuery(string)*, *String getPrediction()* and *Iterator getPrediction(int)*. We omit presenting the set and get methods for the members.

4.1.1 void currentQuery(string query)

In the following we enumerate the main processing steps invoked by a current request.

1. If the current object has never been requested before, then a new variable is added to the network and the variable id and object id are stored in the two hash maps. If the current object is already in the network, then its variable id is retrieved from the hash map *oIDvariableID*. In both cases we now have a variable id representing the currently accessed object.

2. The current variable is made dependent on one of the variables stored in the ring buffer *ancestorIDs*. If the value of *dependencyDecay* is zero, the parent is the immediate predecessor in the sequence of queries. If the value is greater than zero a parent is chosen at random from all variables contained in the buffer (see description of RingBuffer member in section 4.1). If the chosen parent does not create a directed cycle in the network the current variable is connected to its new parent. If this would create a directed cycle, another parent is chosen. This is repeated until a suitable parent has been found. Then, the current variable id is added to *ancestorIDs*.

3. The state of the current object is set to accessed (set as evidence). The CPDFs of all variables are updated. This possibly causes decay of the values of variables.

4. If $numberOfQueriesTillCut > 0$, $currentNumberOfQueries$ is increased by one. If $currentNumberOfQueries > numberOfQueriesTillCut$ (forced cut), the method *closeSession()* is invoked.

4.1.2 String getPrediction() and Iterator getPrediction(int n)

First of all, this method initiates a loopy belief propagation in the Bayesian network. As a result, every variable has a CPDF denoting its access probability given all *valueDecayDelay* previously accessed objects.

There are two versions of this method. One returns a single object id of the corresponding variable id with the highest CPDF value and the other returns an Iterator pointing to the beginning of a vector containing the n object ids with the highest CPDF values. Both of these methods can easily be extended to return the least probable variable(s) instead.

4.2 Bayesian Network

Figure 11 shows the UML diagram for the belief network class. This class implements a Bayesian network. The constructor takes one argument denoting the number of values a variable can take up. For prefetching, the variables take the values 0 or 1 and hence the number of values is 2. Since the Bayesian network should not be restricted to the use of prefetching, we enable this number to be given upon construction of an instance of this class.

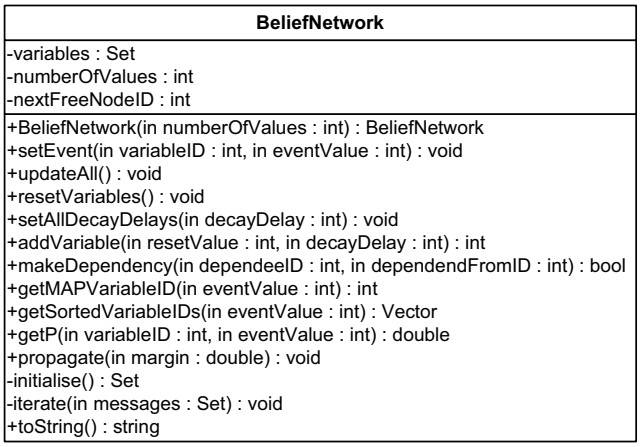

Figure 11: The Belief Network Class

A Bayesian network is defined by its structure and parameters. We implemented a class *Node* which represents a variable. Every instance of this class has a set of nodes that are its parents. It also has the CPDF which is an array of type *float*. The belief network class has a set of this type of nodes which contains all nodes that are in the network. Another member is the integer variable *nextFreeNodeID*. Each time a new variable is added to the network its id is the value of *nextFreeNodeID* and *nextFreeNodeID* is incremented. This guarantees that every variable id is unique. For the purpose of this work we have not implemented the functionality to delete a variable from the network.

Coming up we give a brief description of the member methods:

- *void setEvent(int variableID, int eventValue)* sets the value of the variable referred to by its id to $eventValue$. This is used to set evidence.
- *void updateAll()* invokes an update of the CPDF of every variable in the network.
- *void resetVariables()* sets the value of all variables to $resetValue$.
- *int addVariable(int resetValue, int decayDelay)* creates a new node and adds it to the set of all variables. The integer $resetValue$ is the value to which a node will be set to when *resetValues()* is invoked. This value is also used in belief propagation to determine whether a variable is considered as evidence or not. A variable is evidence if its value is different than $resetValue$. The other argument *decayDelay* is the number of times *update()* can be called on the node before its value is reset to $resetValue$ automatically.
- *boolean makeDependency(int dependeeID, int dependendFromID)* creates a dependency of the variable denoted to $dependeeID$ from the variable referred to by $dependendFromID$. If this connection does not create a directed cycle in the network, then the latter is added to the set of parents of the former and true is returned. If it does create a cycle then this invocation has no effect on the network and false is returned.

- *int getMAPVariableID(int eventValue)* returns the id of the variable with the highest probability of taking the value $eventValue$ given the values of the evidential variables (i.e. $argmax_X\ P(X = eventValue|E = e)$ or, for our prefetcher: $argmax_i\ P(o^i|o_{0..t})$).

- *vector getSortedVariableIDs(int eventValue)* returns a vector containing all variables in the network, sorted according the values of their CPDF $P(X = eventValue|E = e)$ (or, four our prefetcher: $P(o^i|o_{0..t})$). This method is used to retrieve the k most or least probable variables.

- *propagate(double margin)* initiates a loopy belief propagation. It invokes the method *Set initialise()* which returns a set of λ and ρ messages and values to be computed including all messages and values each depends on. This set is passed to *void iterate(Set messages)* which iterates through the list computing messages until all messages are computed. Then all CPDFs $P(o^i|o_{0..t})$ are computed using the λ and ρ values. If no CPDF has changed by a value greater than $margin$ then the propagation stops. In any other case *iterate()* is invoked again and the CPDFs are recomputed until the terminating condition is established.

4.3 Belief Propagation

In section 3.2.3 we have shown that loopy belief propagation is realised with message passing. Every node computes values as a function of messages received from its parents and children. Once it has computed these values it computes the messages it sends to its parents and children and the values of the CPDF $P(o^i|o_{0..t})$.

We have implemented a class *Message* from which the λ and ρ messages as well as the λ and ρ values are derived. The message class defines an abstract method *void compute()* which computes its value as a function of the messages it depends on. Therefore, the message class also defines a protected member set of messages on which its computation depends. The method *Set initialize()* instantiates all messages and values, sets the dependencies and returns a sorted set of the messages and values that have to be computed. The set is sorted in the sense that the iterator that is returned by the *Iterator getIterator()* method defined in the *Set* class starts with the message that has the least messages it depends on and ends at the message that contains the most dependencies.

The method *void iterate()* defined for the class *BeliefNetwork* iterates through this set of messages and calls *compute()* on each. The method *propagate()* now computes the CPDFs $P(o^i|o_{0..t})$ and checks for the termination condition. If it is not fulfilled, *iterate()* is invoked again. If it is fulfilled the propagation comes to an end.

4.4 Graphical User Interface

The graphical user interface (GUI) allows to interact with the proxy server. It has been implemented in [17] and we added two panels specific to our prefetcher.

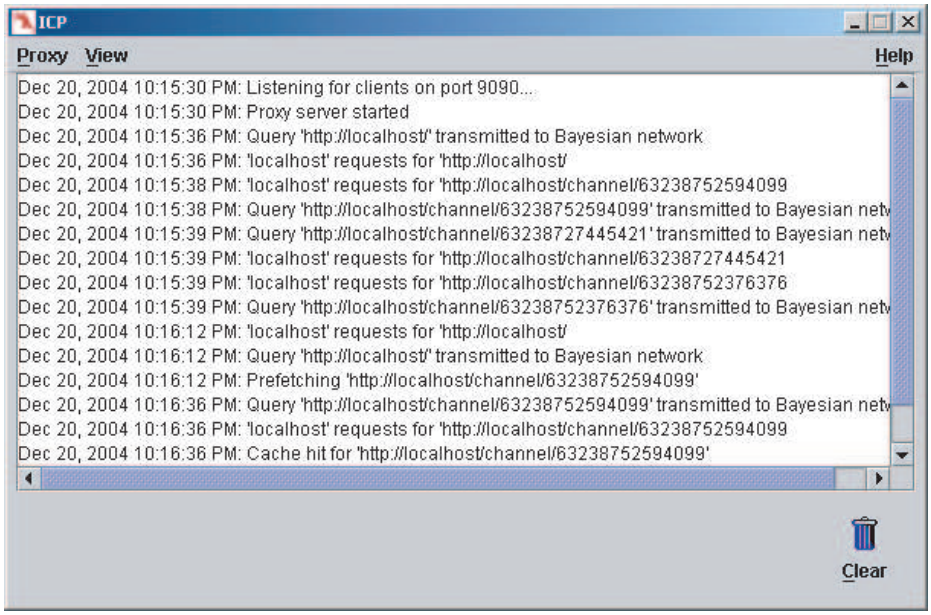

Figure 12: The main view of the graphical user interface

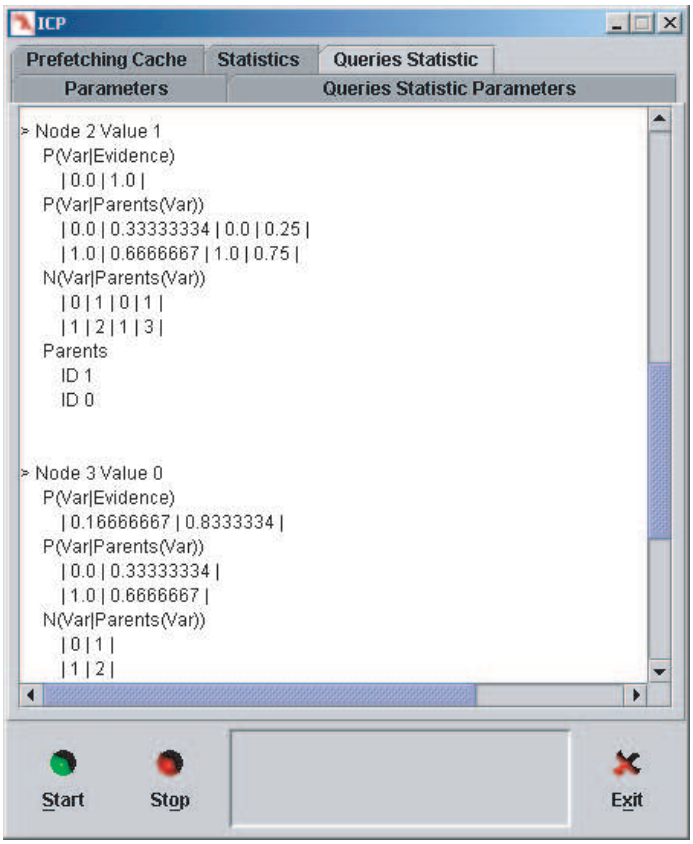

Figure 13: The view on the Bayesian network

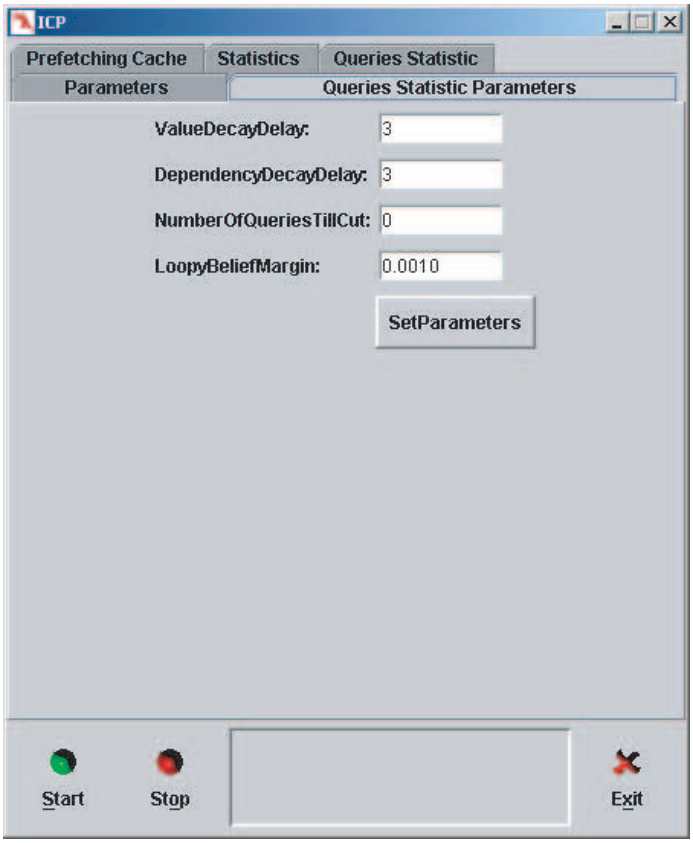

Figure 14: The view to edit the parameters of the Bayesian network

Figure 12 shows the main view of the proxy server GUI. The text area shows all the requests and informs about cache hits and prefetches.

Figure 13 shows the view on the Bayesian network. The text area shows the network in terms of all the variables o^i and their connections. A variable is represented by its id, the counters (introduced in section 3.1), the CPDF $P(o^i|parents(o^i))$ and $P(o^i|o_{1..t})$ as well as an enumeration of its parents. So far the network is printed in ASCII symbols only.

Figure 14 shows the view on the parameters of the Bayesian network (presented in section 4.1). This panels allows to edit and set these parameters.

5 Application and Results

In order to evaluate the performance of our prefetcher we decided to test it first in an artificial environment. Testing it in a sandbox, as opposed to real world application, simplifies the analysis of the functioning. Unfortunately our prefetcher failed to behave as expected, which prohibits real world testing. In this section we describe the set up of our experiments and point out possible causes of failures. In the next section 6 we propose revisions that might lead to better results.

5.1 Sandbox Testing

The original idea for sandbox testing arose from the conclusion that the HTTP protocol does not comply with the standard request response model of conventional information systems. [14] proposes two models of control flow in the interaction of user and information systems. Traditionally, the user requests for information which triggers a response from the system (including side effects which may not be perceivable by the user). In a second model, response may additionally be triggered within the system itself or by other users . Hence, information may be delivered without explicit request, a fact that accounts for the latest trends in information engineering.

Although the HTTP protocol can be regarded as an instance of a request response model, we want to point out another aspect of model classification. The request to an information system usually causes a single response, whereas an HTTP request may implicitly entail further requests and, thus, multiple responses follow the original single request.A website for example can possibly contain files such as style sheets, pictures and other medias which the browser requests independently. If we want to classify the user requests in order to perform usage prediction it is important to differentiate between the request posted by the user and the succeeding requests entailed by the first response. Since the set of objects required to respond to one request can easily be determined without the need for probabilistic inference we avoid burdening the belief network with implicit knowledge. Furthermore, it would complicate evaluating the correct recognition of object clusters arising from usage statistics which becomes clear in the next paragraph.

We use the RSS feed reader [6] which comes along with a HTTP server. The server responds with an HTML site presenting the available RSS feeds as a Menu when queried on top level. Clicking on a menu item requests the respective RSS feed which typically consists of a short text. Thus one request does not imply further requests and only triggers one single response. We selected three sets of RSS feeds each representing a specific area of interest (cluster). If we would allow implicit additional requests we would have to assign the complete set of responses to the clusters and thus unnecessarily classify more objects.

Table 1 enumerates the objects we have selected as requestable from the RSS feed server including the cluster they are assigned to. The numbers in brackets label each item used to shorten references in the tex.

The user interacts with the server by first requesting the overviewing top level

Cluster	Newscast	IT
Members	BBC News, News Front Page (1)	D-INFK News (7)
	Reuters, Top News (2)	D-INFK Events (8)
	BBC News, World (3)	D-INFK Colloquia (9)
	Reuters: World (4)	Wired News: Top Stories (10))
	Reuters: Science (5)	Slashdot (11)
	BBC News, Science/Nature (6)	Ars Technica PC News (12)

Table 1: Clusters of requestable objects and their members

value decay	7	3	3	5	5
dependency decay	0	3	5	3	5
forced cut	6	0	0	0	0

Table 2: Tested Combinations of Parameters

site. From there each RSS feed can be accessed with one click. Note that the clusters proposed in table 1 are unknown to our prefetcher and are supposed to be reflected with the CPDFs learned by the belief network. The required structure and parameters of the network are learned based on usage statistics only.

We simulate a user that sequentially pursues his/her interest by repeatedly clicking on items within a particular cluster before changing interest, and thus viewing items of another cluster. An example sequence of requests could be $1, 2, 5, 4, 6, 3|9, 8, 10, 7, 12, 12$ where the user was first interested in newscast and then changed to IT news.

5.2 Test Set Up and Results

We have tested out prefetcher using various combinations of parameters and access sequences. Table 2 summarizes the tested combinations of parameters. Each column corresponds to a test set up. In a first test we have initiated a forced cut after every 6th request. After each cut all values are reset, therefore the value decay of 7 has no effect. The dependency decay value 0 entails that every request is made dependent on its immediate predecessor only. All other columns test the regular set up of our prefetcher where the values decay automatically and a predecessor is chosen randomly within the *dependencyDecay* preceding requests.

We implemented a test class that simulates a user accessing the RSS items. For each combination of parameters four different access sequences were simulated (the string in italic is used as reference in figures 15 - 23):

- *3 x all, seq.* The user accesses three times the members of the first two clusters in a fixed sequence. The sequence is $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$. Note that there is no differentiation between the two clusters.

- *3 x each, seq.* The user accesses all members of each cluster three times before changing its interest (cluster). The members within a cluster are

always accessed in a fixed order. The sequence of accesses is three times $1, 2, 3, 4, 5, 6$ and then three times $7, 8, 9, 10, 11, 12$.

- *3 x all, rnd.* The user accesses three times the members of the first two clusters. As opposed to the first case there is no fixed order of access - the sequence of access is random. Note that there is no differentiation between the clusters.

- *3 x each, rnd.* The user accesses all members of each clusters three times before changing its interest. The members of a cluster are accessed in random order.

The cases where the user requests all items as if they belong to a single cluster serve as a base to evaluate the effect of the clustering in the two other cases. The functioning of our prefetcher would manifest itself if there was a difference between accessing all items in a row and accessing the members of each cluster repeatedly before changing the cluster.

We also tested the effect of always keeping the root node as a potential predecessor as opposed to treating it as a regular predecessor and dropping it after *dependencyDecay* requests. This set up yields 36 tests. For each test we have run the user simulation 20 times before evaluating the prefetcher with the following access sequences: The members of each cluster are accessed 10 times either in a fixed order or randomly (corresponding to the respective test set up). Since we test using two clusters only, the evaluation is based on 20 access sequences: the first 10 sequences access members of the first cluster and the second the ones from the second cluster. For each sequence we count the number of hits. A hit within a sequence accessing a particular cluster occurs when our prefetcher proposes to prefetch a member of this cluster. Figures 15 - 23 show the results of each test. We have drawn a black vertical line to differentiate the sequences accessing the first cluster (to the left of the line) from the ones accessing the second cluster (to the right of the line).

In most test cases the hit rate for the sequences accessing the first cluster is acceptable. But the accesses to the members of the second cluster do not show a better hit rate than the cases where the prefetcher has been trained with accesses not specific to any cluster. In those cases where the hit rates are similar within both clusters (figures 17 and 19) the rates are low for both of them. The only case where the prefetcher performs more or less successfully is where we use a forced cut according to the number of members of the clusters (for random access sequences only, figure 15). Since we do not want to incorporate knowledge about session endings this set up is not an option for real world application.

Changing the values of the parameters produced little improvement in only one case: Always using the root node as a potential predecessor in combination with increased value and dependency decay (5 each) produced the best results only for random access sequences (figure 23).

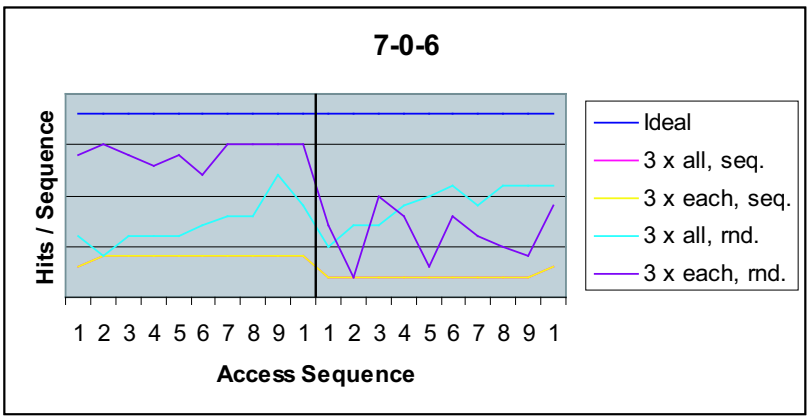

Figure 15: Four tests for evaluating the prefetcher using the forced cut after every 6th request. The three numbers on top denote a row in table 2

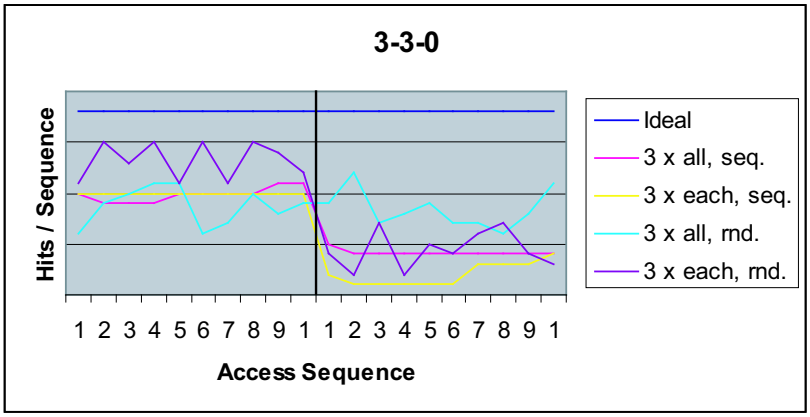

Figure 16: Four tests for evaluating the prefetcher with a value decay after 3 requests and a dependency decay value equal to 3. The root node is treated like any other node.

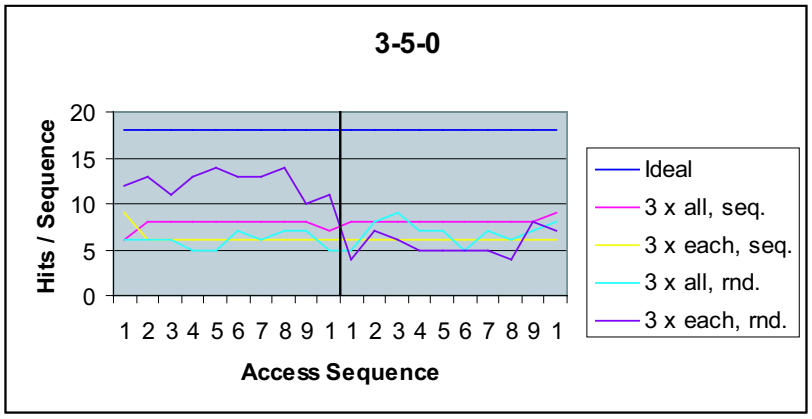

Figure 17: Four tests for evaluating the prefetcher with a value decay after 3 requests and a dependency decay value equal to 5. The root node is treated like any other node.

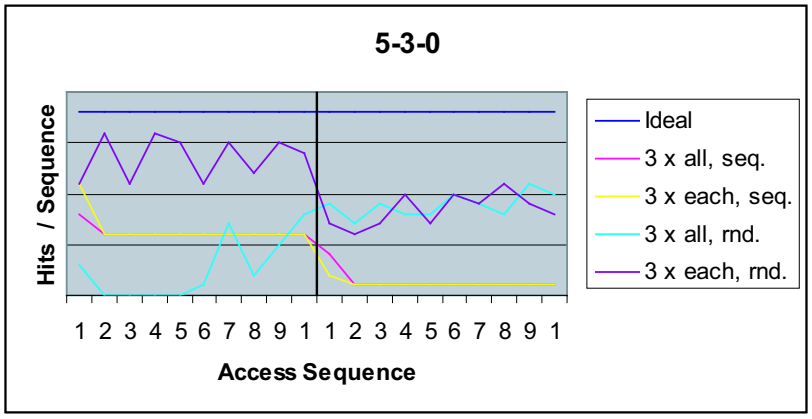

Figure 18: Four tests for evaluating the prefetcher with a value decay after 5 requests and a dependency decay value equal to 3. The root node is treated like any other node.

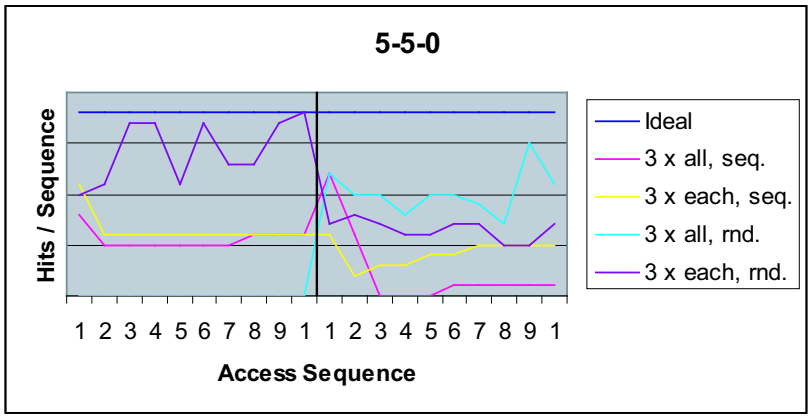

Figure 19: Four tests for evaluating the prefetcher with a value decay after 5 requests and a dependency decay value equal to 5. The root node is treated like any other node.

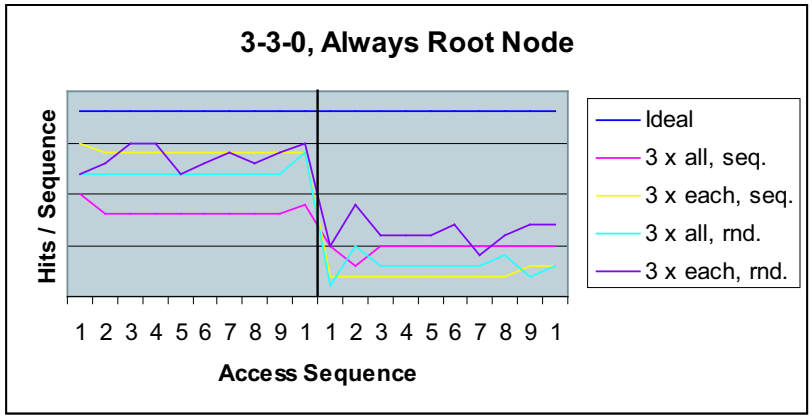

Figure 20: Four tests for evaluating the prefetcher with a value decay after 3 requests and a dependency decay value equal to 3. The root node is always kept as a potential predecessor.

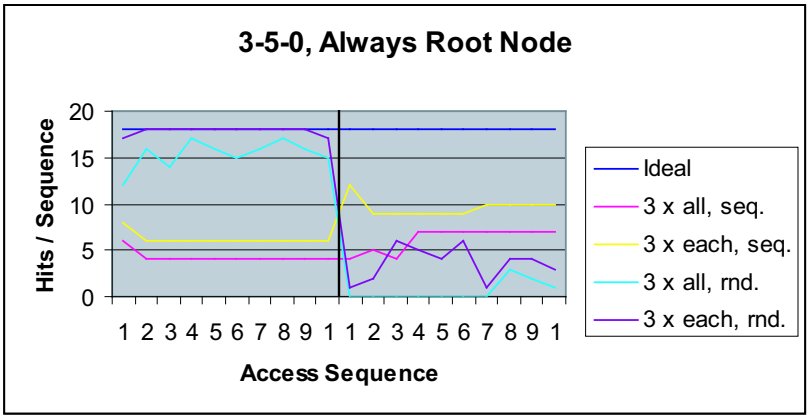

Figure 21: Four tests for evaluating the prefetcher with a value decay after 3 requests and a dependency decay value equal to 5. The root node is always kept as a potential predecessor.

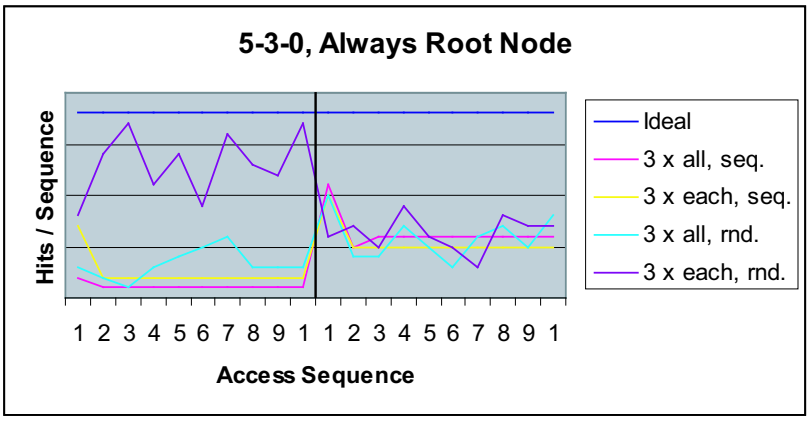

Figure 22: Four tests for evaluating the prefetcher with a value decay after 5 requests and a dependency decay value equal to 3. The root node is always kept as a potential predecessor.

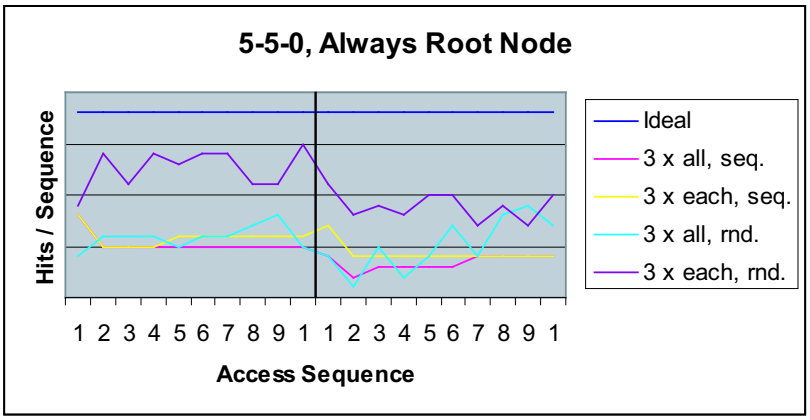

Figure 23: Four tests for evaluating the prefetcher with a value decay after 5 requests and a dependency decay value equal to 5. The root node is always a potential predecessor.

5.3 Possible Origins of Failure

We are convinced that the JPDF modeling the accesses over all objects is a powerful help for access prediction. Thus we believe that the problem lies in the correct approximation of the JPDF. Following we enumerate possible causes of incorrect JPDF approximation.

- Firstly, the belief propagation algorithm we applied produces an approximation only of the CPDFs and, secondly, it may fail to converge towards the correct CPDF values.
- The dependencies built up during the usage of the HTTP proxy server do not necessarily capture the true dependencies. The assumption that *the properness of the connections are not vital to the correct computation of the CPDFs since dependencies arise between variables that are not connected* might be wrong.
- We use a value and dependency decay to avoid the necessity of knowledge about session endings. If the dependencies modeled in the belief network are vital to correct inference our predecessor selection mechanism might fail to capture the true dependencies. The value decay might not correctly substitute the value reset at the end of a session if the end was known.

6 Future Work

In this section we propose future work in order to make our prefetcher work correctly on one hand and to add desirable functionality on the other hand. Since our tests show that the current state of implementation does not function correctly it is important to first prove whether a belief network actually works for access prediction or not.

Following we enumerate improvements that may enable a proper functioning of our prefetcher.

- Loopy belief propagation is an approximate inference mechanism that might fail to converge towards the correct CPDF values. In order to find out if the propagation algorithm is responsible for the failure we propose to implement either an exact belief propagation algorithm or an approximative algorithm that does not possibly converge to malicious values. Since the exact inference is NP-complete such an algorithm might be useless for real world application. Its sole utility would be to proof whether the propagation is or is not responsible for the failure.

- Since we assumed that the dependencies modeled in a belief network do not necessarily need to capture the true dependencies it is important to investigate the importance of connections for the correct representation of the JPDF within the network.

- An obvious approach to increase the correctness of the dependencies represented in the belief network is to periodically analyse the network. Such an analysis could propose connections to be added or erased. The analysis would be based on the local CPDFs that are computed by counting during the usage of the HTTP proxy server ($P(o^i|parents(o^i))$). For every parent of a variable we could compute a dependency index capturing the covariance between the two. If the variables do not seem to be dependent the connection could be removed from the network.

- So far, the parameters to the network ($valueDecay$, $dependencyDecay$ and $forcedCut$) were predefined. These parameters could as well be appropriated and adapted during the usage of the proxy server. The PDF for the selection of a predecessor could be converted into a parameter in order to incorporate additional information available to the prefetcher (e.g. the more time elapses between two requests the higher the probability that the second request is connected to the root node).

- The parameter $valueDecay$ could be replaced by two value decay parameters. One could be used as before but only for the empirical learning of the CPDFs. The other could be used for the belief propagation. Therefore, a second ring buffer would be used to store the previous accesses. Every time belief propagation is performed all values are reset and only the ones from the second ring buffer are set as evidential. The effect of maintaining a different set of evidential variables for parameter learning and belief propagation would have to be tested.

- If it can be proven that the belief propagation algorithm is not responsible for the failure, we could try to recognise session delimiters. This would require to incorporate further information into the prefetcher and hence to expand its interface. Additional information available to the prefetcher can be extracted by parsing the URL of the requested objects, by parsing the information contained in the objects and by interpreting temporal properties of accesses.

In order to extend the functionality of our prefetcher we propose the following extensions:

- For the purpose of this work we have not implemented the ability to erase variables in the belief network. For the purpose of maintaining the network and keeping its size bound this would be a desirable functionality. Using this ability the network could be analysed periodically to find variables with low prior probability and to erase them from the network.

- The graphical user interface of the HTTP proxy server displays the belief network only numerically. The representation of the network could be enhanced to graphically display the graph structure in terms of the nodes and edges.

- The most important additional functionality would be to make use of further inference abilities of a Bayesian network presented in appendix 8. Inference could be used for schema evolution. A schema defines relationships within the set of accessed objects in terms of associations and properties. Associations arising from schema evolution can be used to further improve access prediction and to feed back information into the information system.

7 Appendix: Bayesian Probability Theory

7.1 Probability Distribution Functions

Probabilistic reasoning is mainly based on Bayesian probability theory. The foundation of Bayesian probability theory is the famous Bayesian equality:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{8}$$

$P(A)$ is a probability distribution (PD) for a random variable A. $P(A)$ contains an entry for any state that A can be in, denoting the probability of the event that A is in that state. In the case that A is a binary random variable A can take the value 1 with the probability $P(a = 1)$ and 0 with the probability $P(a = 0) = 1 - P(a = 1)$. A coin toss is a common used example for a binary random variable where $P(coin = head) = 1 - P(coin = tail) = \frac{1}{2}$ if the coin is fair.

$P(A|B)$ is a conditional probability distribution (CPD) for a random variable A given the state of another random variable B. $P(A|B)$ is a table that contains an entry for any possible combination of the states of A and B denoting the probability of the event that A is in a state given that B is in a particular state.

For example let us assume that A and B are binary random variables. $P(A|B)$ contains the probabilities $P(a = 0|b = 0)$, $P(a = 0|b = 1)$, $P(a = 1|b = 0)$ and $P(a = 1|b = 1)$. To illustrate this example using the coin toss from above, let us assign to B the two possible states 'Our coin is fair' and 'Our coin is unfair'. A is the outcome of the tossing and can take the value 'Head' or 'Tail'. Now the meaning of a CPD should become clear, since we obviously want to differentiate $P(a = head|b = coin\ is\ fair) = \frac{1}{2}$ from $P(a = head|b = coin\ is\ unfair) \neq \frac{1}{2}$.

Equation 8 is the foundation of probabilistic inference and reasoning but it is irrelevant to this work to introduce its various applications. Nevertheless, there is one more PD that we want to introduce:

$$P(A, B) = P(A|B)P(B) \tag{9}$$

$P(A, B)$ is called the joint probability distribution (JPD) over the random variables A and B. Again, $P(A, B)$ is a table containing entries that denote the probability of an event consisting of two particular states that A and B are in. In contrast to the CPD $P(A|B)$, where an entry denotes the probability of A being in a certain state *given* that B is also in a certain state, the JPD $P(A, B)$ denotes the probability that A is in a certain state *and* B is in a certain state.

To continue our coin example we assume A to be the outcome of a tossing and B the fact that our coin is or is not fair. If we know $P(A|B)$ and $P(B)$ we can easily compute the JPD as $P(a = i, b = j) = P(a = i|b = j)P(b = j)$ with $i, j = \{0, 1\}$[1]. The JPD tables is the computation for any combination of i and j.

$P(A, B)$ can also be computed as $P(B|A)P(A)$ which allows us to derive Equation 8 using $P(A|B)P(B) = P(B|A)P(A)$.

The JPD can be regarded as an overall PD since it contains all the information about any PD over any combination of random variables contained in the JPD. We can compute the PD $P(A)$ from $P(A, B)$ by 'marginalisation over B':

$$P(A) = \sum_B P(A, B) \tag{10}$$

Now we can compute $P(B|A) = \frac{P(A,B)}{P(A)}$ and vice versa for $P(A|B)$. Thus, if we know the JPD over all random variables of interest, we can compute any PD that is required for any kind of probabilistic reasoning. On the other hand, and this is of particular interest for belief networks, we can compute a JPD using CPDs and PDs.

The last concept we want to introduce is the conditional independence. A conditional independence statement (CIS) written as $CIS(A, B|Z)$is said to hold if A and B are independent given Z. That is, $P(A|B, Z) = P(A|Z)$ and $P(A, B|Z) = P(A|Z)P(B|Z)$. The main contribution of belief networks is making it possible to take advantage of dependency structures governing over a set of random variables. Independencies can easily be read of and used to compute a JPD with as less computational effort as possible. As a very simple example, the equation $P(A, B, C) = P(A)P(B|A)P(C|A, B)$ simplifies to $P(A, B, C) = P(A)P(B|A)P(C|A) \iff CIS(B, C|A)$ holds.

7.2 Empirical approximation of PDFs

In this section we briefly describe how to approximate the PDFs using observations. The incremental approximation of the CPDF constitutes the parameter learning of our belief network implementation. We use the notation N_{Global} to denote the total number of observations. n_P is the number of observations satisfying the condition P. Note that A and B can also be a set of random variables which does not affect the formulas.

$$P(A) = \forall a \in A : \frac{n_{A=a}}{N_{Global}} \tag{11}$$

$$P(A|B) = \forall a, b \in A, B : \frac{n_{A=a\&B=b}}{n_{B=b}} \tag{12}$$

$$P(A, B) = \forall a, b \in A, B : \frac{n_{A=a\&B=b}}{N_{Global}} \tag{13}$$

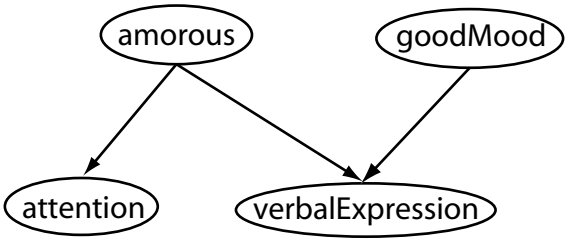

Figure 24: Example Bayes Network with four Variables

$P(attention\|amorous)$	$amorous = false$	$amorous = true$
$attention = poor$	0.2	0.9
$attention = rich$	0.8	0.1

Table 3: The CPDF $P(attention|amorous)$

8 Appendix: Bayesian Networks

A Bayesian network is a connected and directed acyclic graph that represents random variables and their dependencies. A variable may be observable (e.g. verbal expression of a person) or unobservable (e.g. mood of a person). A variable is said to depend on another if the latter causes the former (e.g. bad mood causes aggressive verbal expression). In the network such a dependency is represented by a directed edge pointing from the cause to the effect. The cause is often referred to as the parent of the effect.

The general idea of Bayesian networks is to model the JPDF over all variables contained in the network in terms of local CPDFs. As we have derived in appendix 7, the JPDF contains all information, i.e. all possible PDFs over any subset of the variables. The computation of a CPDF by marginalisation over the JPDF is expensive ($\in O(2^n)$). The Bayesian networks allows the more efficient computation of a CPDF by considering independencies modeled in the network [15].

A network is defined by the structure (i.e. the nodes and the causal connections from its parents) and the parameters, i.e. a CPDF for every node with parents ($P(node_i|parents(node_i))$ with $parents(node_i)$ denoting the set of parents of $node_i$) and a prior PDF for every node without parents ($P(node_i)$).

Once the network is set up, we can infer knowledge. Typically, a subset of the variables is set to values that have been observed. The observed variables are referred to as evidential variables or simply evidence. The values of interest are the CPDFs $P(v|evidentialvariables)$ of all the non evidential variables v. Using belief propagation as proposed in [15] these CPDFs are computed directly using message passing between the nodes but without explicitly computing the JPDF and marginalisation.

We present a simple example network in order to point out the basic functioning of Bayesian network. We assume that a person is either in love or in a good

$P(verbExp\|amorous, goodMood)$	$am = false$ $gM = false$	$am = false$ $gM = true$	$am = true$ $gM = false$	$am = true$ $gM = true$
$verbExp = normal$	0.7	0.4	0.2	0.1
$verbExp = friendly$	0.3	0.6	0.8	0.9

Table 4: The CPDF $P(verbalExpression|amorous, goodMood)$

mood. Both states are modeled using two binary random variables: $amorous \in \{false, true\}$ and $goodMood \in \{false, true\}$. Further variables are $attention \in \{poor, rich\}$ and $verbalExpression \in \{normal, friendly\}$. The attention is modeled to depend on the amorous state only, i.e. if the person is in love he pays poor attention and vice versa. We further assume that the verbal expression depends on both, the amorous state and the good mood. If the person is in a good mood, his verbal expression is friendly and if the person is in love, the expression is rather friendly as well.

So far we have defined the structure of the network in terms of it's nodes and dependencies. Now we need to define the parameters: The tables 3 and 4 define the CPDFs of the nodes with parents, i.e. for the variables *attention* and *verbalExpression*. The variables *amorous* and *goodMood* require prior probability distributions. Let us define them as $P(amourous = true) = 0.2$ and $P(goodMood = true) = 0.7$. Picture 24 shows the network. All these values could be either set by an expert or retrieved in data by counting the corresponding events and using the equations 11 and 12 defined in appendix 7. In our example we have simply used some common sense.

Now that the network is fully defined, we can start collecting evidence and run belief propagation to infer the CPDFs $P(x|evidence)\ \forall$ *nodes* x each time new evidence arises. A typical inference could be that $P(amorous|attention = poor) > P(amorous)$, in other words, the fact (evidence) that a person pays poor attention increases the probability that this person is in love. There are three main probabilistic inference tasks for which a belief network is commonly used. All share the use of CPDFs in the form of $P(X|E)$ where E is the set of evidential variables and X is the set of all non evidential variables.

- *Belief Assessment* is a query for $P(X_i = x_{ij}|E)$. $P(X_i = x_{ij})$ is the probability that variable i has the jth value of its vector of possible values. In our example such a query could be verbalised as *what is the probability that a person is in love given that he/she exhibits friendly verbal expression.*

- *Most Probable Explanation (MPE)* seeks for the most probable values of all non evidential variables given the evidence. This query can be formulated as $argmax_x\ P(X = x|E)$. In our example this query translates into *what are the most probable values for the variables amorous, attention and goodMood given a friendly verbal expression.*

- *Maximum a Posteriori (MAP)* inference is similar to MPE. The difference is that now we are interested in the most probable values of a *subset* of all non

evidential variables. The formal query is $argmax_u\ P(U = u|E)$ where U is a subset of all non evidential variables X. If U contains one variable only, then the respective MAP inference corresponds to a Belief Assessment.

There are some properties particular to Bayesian networks we want to point out. In traditional logic, the implication $a \implies b$ does not allow the reverse implication $b \implies a$, hence, if b is observed we know nothing about a. In Bayesian networks, if we have the dependency $goodMood \implies friendlyVerbalExpression$ and we observe a friendly verbal expression, the probability of a good mood increases. This is closer to human reasoning.

If a friendly verbal expression is observed, the probability of the person being in a good mood and being in love increases equally. Since the probability of the person being in love has increased, the probability of the person paying poor attention increases as well. Hence, the observation of friendly verbal expression increases the probability of poor attention. These two variables are dependent from each other although there is no direct dependency denoted by an edge.

The third particularity is often referred to as explaining away. We observe that the person has a friendly verbal expression. As stated before, this increases the chances of the person being in a good mood as well as the person being in love. If we additionally observe that the person pays poor attention, we know that there is love. Since this fact explains the person's poor attention, we can drop the explanation using the persons good mood. The fact that the person is in love explains away that he is in a good mood and hence the probability of good mood decreases. Again, two variables exercise influence on each other although they are not explicitly modeled to do so by a dependency edge.

9 Appendix: API

References

[1] Bouras, C. and Konidaris, A., Kostoulas, D. *Predictive Prefetching on the Web and its Potential Impact in the Wide Area* World Wide Web: Internet and Web Information Systems, 7, 143-179, 2004.

[2] Charniak, E. *Bayesian Networks without Tears* The American Association for Artificial Intelligence, Winter 1991.

[3] Chen, Z., Lin, F., Liu, H., Liu, Y., Ma, W., Wenyin, L.*User Intention Modeling in Web Applications Using Data Mining* World Wide Web: Internet and Web Information Systems, 5, 181 - 191, 2002.

[4] Curewitz, K., Krishnan, P., Vitter, J. *Practical Prefetching via Data Compression* SIGMOD, 1993

[5] Davison, B. *Learning Web Request Patterns* In Poulovassilis, A. & Levenen M. *Web Dynamics: Adapting to Change in Content, Size, Topology and Use* pp. 435-460, Springer, 2004.

[6] Feedreader: *A Lightweight Open-Source Aggregator that Supports RSS and ATOM Formats* Version 2.7 646, http://www.feedreader.com, 2004.

[7] Getoor, L., Friedman, N., Koller, D., Taskar, B. *Learning Probabilistic Models of Link Structure* The Journal of Machine Learning Research, Volume 3, Pages: 679 - 707, March 2003.

[8] Getoor, L. Segal, E. Taskbar, B. Koller, D. *Probabilistic Models of Text and Link Structure for Hypertext Classification* In Proc. IJCAI01 Workshop on Text Learning: Beyond Supervision, 2001.

[9] Horvitz, E., Breese, J., Heckerman, D., Hovel, D., and Rommelse, K. The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users. Proceedings of UAI 1998

[10] Jiang, M. *Mathematical Models in Computer Vision and Image Processing* Department of Information Science, School of Mathematics, Peking University, July 2001.

[11] Jaronski W., Bloemer, J., Vanhoof, K. and Wets, G. *Use of Bayesian Belief Networks to Help Understand Online Audience* Proc. Data Mining Marketing Appl. Workshop ECML/PKDD 2001. Freiburg, Germany.

[12] Murphy, K., Weiss, Y., Jordan, M. *Loopy Belief Propagation for Approximate Inference: An Empirical Study* In Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence, pages 467-475, Sanfransisco, CA, 1999.

[13] Norrie, M.C. and Würgler, A. and Palinginis, A. and von Gunten, K. and Grossniklaus, M. *OMS Pro 2.0 Introductory Tutorial* Inst. for Information Systems, ETH Zurich, March 2003.

[14] No Author Given. *An Infrastructure for Reactive Information Environments* Inst. for Information Systems, ETH Zurich, 2004.

[15] Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[16] Shwe, M., Middleton, B., Heckerman, D., Henrion, M., Horvitz, E., Lehmann, H., Cooper, G. *Probabilistic diagnosis using a reformulation of the INTERNIST 1/QMR knowledge base. I. The probabilistic model and inference algorithms* Methods Inf Med 30 (In 1993): 241-55, 1991.

[17] Signer, B., Erni, A., and Norrie, M.C. *A Personal Assistant for Web Database Caching* Proceedings of CAiSE'2000, 12th International Conference on Advanced Information Systems Engineering, Stockholm, Sweden, June 2000.

[18] de Spindler, A. *Pearl's Polytree Algorithm* Seminar on Belief Propagation, ETH Zurich, Switzerland, Winter 2003.

[19] Wang, Y. *Web Mining and Knowledge Discovery of Usage Patterns* February 2000.