An MDP-based Peer-to-Peer Search Server Network

Yipeng Shen Dik Lun Lee
Department of Computer Science
Hong Kong University of Science and Technology
Hong Kong, China
{yipeng, dlee}@cs.ust.hk

Abstract

A distributed search system consists of a large number of autonomous search servers logically connected in a peer-to-peer network. Each search server maintains a local index of a collection of documents available at the server or on other peer machines. When a query is received by any server in the network, a distributed search process determines the most relevant search servers and redirects the query to them for processing.

In this paper, we model the distributed search process as Markov Decision Processes (MDPs). The estimated relevance of a server to a query is regarded as the reward in the MDP model. Once the MDP policies representing the global knowledge are obtained at each server through asynchronous value iteration, the most relevant servers to a given query can be efficiently identified despite the lack of centralized control and global knowledge at each autonomous server.

We discuss the implementation and complexity of the asynchronous value iteration and how we extend the traditional MDP to handle the multiple-access policy (i.e., more than one optimal server is returned) and queries with multiple terms. Finally, experiments are conducted using the TREC collection. We show that the MDP-based distributed search can achieve results very close to that of a centralized search.

1 Introduction

The Web is a distributed environment where various kinds of resources are massively distributed and loosely connected by hyperlinks. A centralized search engine is not able to adapt to the dynamic environment in terms of the ability to cover the entire Web and to refresh the index frequently. To alleviate these problems, distributed search systems [10, 13, 8] delegate the task of indexing the huge and ever-changing web contents to a large number of autonomous search engines[1]. Each of the search engines only indexes a small and different part of the Web. As such, it is able to refresh its index frequently. The search engines form a distributed search community.

In this paper, we adopt a peer-to-peer distributed search architecture, in which each search engine is autonomous and connected to only a small number of peer search engines. Since the number of servers in the system may be very large, it is very costly to route a query to every server in the system for processing. The resource consumption for each query consists of the network communication cost and the query execution cost in the servers. To utilize the system resource more effectively, the query should be routed only to the most relevant servers in the system for processing. In order to identify the most relevant servers, we model the distributed search process in a flat architecture as Markov Decision Processes (MDPs). The query can be routed to the most relevant server by aiming at obtaining the optimal reward through the optimal routing action (i.e., the MDP policy) computed by the MDP value iteration algorithm.

The MDP policies must be pre-computed in order to provide acceptable response time. This is an analogue to document indexing. While it is practical to compute the MDP policies for single terms because the number of distinct terms in a language is limited, it is impractical to do so for multiple-term combinations. In this paper, we propose methods to answer multiple-term queries based on MDP policies obtained for the single terms. We introduce the asynchronous value iteration algorithm for our MDP model and study its complexity. The estimated relevance of a server to a query term is incorporated in the MDP messages, which are propagated in the server network during the asynchronous value iteration. We extend the traditional MDP to handle the multiple access policy (i.e., more than one optimal server is returned) and queries with multiple

[0]To appear in WISE 2002.

[1]We also refer to a *search engine* as a *search server* or *index server*.

terms. Once the MDP policy is computed in our MDP model, given a query at any server, we can directly identify the most relevant servers in the system. Therefore, the query can be directly sent to the most relevant servers for evaluation without the need to route it from server to server as the logical distributed search model may have implied.

Section 2 reviews the related work in the distributed search area. Section 3 introduces the distributed search system promoted by MDPs. Section 4 provides the asynchronous value iteration algorithm for our MDP model as well as the algorithm's complexity. Section 5 presents the experimental results. We draw our conclusions in Section 6.

2 Related Work

A distributed search is a ranking problem on multiple search servers. Given a query, we need to estimate the relevance of the search servers and forward the query to the most relevant search servers for processing. A distributed search system can be organized in a hierarchical or peer-to-peer architecture.

Hierarchical distributed search systems begin with two layers. *Gloss* (the Glossary-of-Servers Server) has a typical two-layer infrastructure [2]. *gGloss* is a widely adopted server ranking method based on the vector space model [8]. *CORI* (Collection Retrieval Inference Network) is an inference network model for selecting the most relevant servers from a group of servers [4, 10]. *Cue Validity Variance* (CVV) [15] is a server ranking method for two-layer systems and uses document frequency (DF) only. Experiments evaluating these ranking methods, such as $gGloss$ and $CORI$, were reported in [3, 6, 10, 12].

A two-layer architecture can be generalized into a hierarchical system with multiple layers of brokers while the search servers are still maintained at the leaf nodes of the hierarchy [14, 8]. Unlike the flat system architecture, the hierarchical architecture is not likely to scale up for the Internet.

Ingrid [5] is a system with a peer-to-peer architecture. In order to direct the query to the most relevant servers, servers containing the same indexed terms are connected together and form a term cluster. Viles and French [13] discussed the influence of the dissemination of collection-wide-information in a flat server network. JXTA Search [7] focuses on the protocol specification of the communication between the peers in a flat server network.

3 Incorporating an MDP in a Distributed Search System

Markov Decision Processes (MDPs) [11] are concerned with the optimal strategies of a decision maker who must make a sequence of decisions over time. A query proposed to a server in the distributed search system may be routed to the optimal server step by step in the server network. After receiving a query at a server, we have to decide whether to pass the query to a neighbor which is closer to the optimal server or to retrieve the current server since it is already the optimal server. We illustrate how to model the distributed search as an MDP in this section. First, for each query term, the estimated goodness score of a server representing the server's relevance to the term is introduced; this goodness score is regarded as the local immediate reward in the MDP. Furthermore, two other major issues in our MDP model (i.e., multiple access policy and queries with multiple terms) are discussed. Finally, we identify and describe the components of an MDP-based search server.

3.1 The Goodness Score of a Server

Generally, given a query term, if many documents in a server's index database contain the term and this term appears more frequently in these documents, then we believe the server is more relevant to the term. Suppose we have term t and document j, and that $TF_{j,t}$ is the *term frequency* (i.e., the number of occurrences of t in j). $TF_{j,t}$ is further adjusted to $tf_{j,t}$ according to the greatest TF value in document j [1].

$$tf_{j,t} = 0.5 + 0.5 \times TF_{j,t} / \max_{k \in j}(TF_{j,k}).$$

Then we define $G_{i,t,l}$, the goodness score of server s_i with respect to term t, as the sum of tf values of all the documents in server s_i whose tf values are greater than the threshold l:

$$G_{i,t,l} = \sum_{doc_k \in s_i \wedge tf_{k,t} > l} tf_{k,t}. \quad (1)$$

If l is set to zero, $G_{i,t,0}$ is the complete sum of tf values of all the documents containing t in s_i. The goodness score of a server with respect to a query q consisting of multiple terms can be constructed according to the goodness scores for the query terms. This will be discussed in detail in the subsequent subsections. Since the goodness score represents the estimated relevance of the server, it is regarded as the local immediate reward in our MDP model.

3.2 MDPs and Distributed Search

An MDP is a model for sequential decision processes in a dynamic system. It consists of four components (i.e., a set of all possible states, each state's set of possible actions, transition probability, and immediate rewards obtained by the decision maker). The decision maker may choose an action (decision) from a number of alternatives according

to the current state of the system. Once the chosen action is executed, the current state of the system may change and the decision maker receives an immediate reward returned by the action. Acting optimally at each state, the decision maker aims at obtaining the greatest reward after a series of actions.

In a flat distributed search system, each server is connected[2] to several neighbors. Let $S = \{s_1, s_2, ..., s_N\}$ be the set of servers in our system. Given a query, suppose only the most relevant server in S is returned. If we propose a query q to server s, the relevance of s to q is estimated. We start at server s and determine whether to answer the query at this server or to pass the query to a neighbor in the server network. In other words, if we think s is very likely to be the most relevant server, s is returned. If we think another server somewhere in the server network is more likely to be the most relevant server, we deliver q to one of the neighbors of s which is believed to be closer to the optimal server. The same decision procedure is further processed at the server who receives query q.

Thus, the distributed search in a flat server network is actually a sequential decision process and it can be modelled as an MDP. Given a query to any server in the server network, the MDP may route the query to the optimal server according to the precomputed optimal decision. A formal description is given below.

Every server in the distributed system stands for a state in the MDP. The server where query q currently resides can be regarded as the present state of the query processing. S is the set of all possible states. The current state of query processing changes when the query is passed to a neighbor.

There are two kinds of actions at any server s which receives q from the other servers or the user. That is, s may either answer it or pass it to a neighbor. These actions, $A_s = \{a_1, a_2, ...\}$, are the alternatives that the decision maker may choose at server s. If the chosen action a is to answer query q at s, we obtain an *immediate reward* $r_q(s, a = s)$, which is the goodness score of server s representing the relevance of server s to query q. If the action a is to pass the query to a neighbor s', the immediate reward $r_q(s, a = s')$ is 0 and the action takes us to server s'.

Given a query, our aim is to locate the most relevant server in the system. In other words, we want to obtain the greatest relevance value. $V_q(s)$, the optimal value function, is the expected total reward that we can obtain by acting optimally starting from server s. According to Bellman's optimality principle [11], we have the following:

$$V_q(s) = max_{a \in A_s}[r_q(s, a) + \lambda V_q(T(s, a))]. \quad (2)$$

[2]This refers to a logical connection in that the server has *direct* message exchange with only a few other servers.

$T(s, a)$ stands for the next state (server) if action a is chosen. If q is passed to a neighbor (i.e., $a \neq s$), $T(s, a)$ is the neighbor who receives q. Otherwise, $T(s, a)$ (i.e., $a = s$) remains as s. The transition $T(s, a)$ caused by action a is deterministic in our model. The state of the query execution switches to another server with probability 1 if action a is chosen.

The first term in the equation is the local reward received right away, which is zero when the action a is to proceed to another server. The second term is the reward that can be received in the future, which is zero if the action is to remain at the current server.

$r_q(s, a = s)$, the estimated relevance of server s, can be defined as the goodness score of s. It serves as the local immediate reward obtained right away at server s.

λ is a discount factor between 0 and 1. It captures the communication cost between adjacent servers. Future reward is discounted with respect to the current reward because we would prefer nearby servers that are quite good to better servers that lie far away, especially when the communication between servers is strictly confined to the server network. In addition, the computation cost can be reduced without losing much precision if λ is set to a proper value when we compute the optimal policy which leads the query to the optimal server.

Starting from a server s, in order to achieve the expected total reward $V_q(s)$, the decision maker should always act optimally. The optimal policy is the optimal action $d_q(s)$ that the decision maker may select at every server s. The greatest expected reward is obtained if the decision maker chooses the optimal action at every decision time. The optimal policy can be obtained as follows:

$$d_q(s) = arg \ \ max_{a \in A_s}[r_q(s, a) + \lambda V_q(T(s, a))]. \quad (3)$$

The optimal policy may be computed by means of *value iteration* [11]. In our MDP model, the value iteration can be simplified according to its specific Bellman equation where the transition is deterministic. In terms of the MDP set up for query q, $V_{q,0}(s)$ may initially be set to 0 for every s. Then we iteratively compute $V_{q,i}(s)$ $(i = 1, 2, ...)$ with respect to (4) at every server s until the sequence of value functions $V_{q,i}(s)$ $(i = 1, 2, ...)$ converges to the optimal value function $V_q(s)$:

$$V_{q,i}(s) = max_{a \in A_s}[r_q(s, a) + \lambda V_{q,i-1}(T(s, a))]. \quad (4)$$

For a query q posted at s_i, according to the optimal action a_i obtained by value iteration, q may be passed to another server and so on until it reaches a server s_m where the action is to stick to s_m and the query is answered at s_m (i.e., s_m is the optimal server). s_m is called the $origin$ of the optimal reward $V_q(s_i)$ at server s_i.

3.3 Multiple Access Policy

The routing procedure described in the last subsection is single cast: only the most optimal server is selected. To return more relevant servers, the value iteration algorithm must be extended.

This can be achieved by not only keeping the greatest expected reward $V_q(s)$ and the corresponding optimal action, but also memorizing the top-k best expected rewards, the corresponding optimal actions, and their corresponding origins. Every $origin$ should be distinct since k different optimal servers are to be selected. In other words, a top-k list is reserved in the value iteration at every server for query q. The *multiple access policy*, which provides k optimal servers with respect to a query proposed at any server, consists of the top-k lists at all servers.

Formally, every item in the list for query q has three components: the expected total $reward$, the $action$, and the $origin$ of the reward. For example, the following is the mth item in the list at server s:

$$< V_q(s,m),\ a_q(s,m),\ o_q(s,m) >.$$

$o_q(s,m)$ is the $origin$ of the expected reward $V_q(s,m)$ and $a_q(s,m)$ represents the action approaching the origin $o_q(s,m)$.

At every server s, we reserve the following top-k list containing k different optimal $origin$s for query q:

$$\begin{array}{c} < V_q(s,1),\ a_q(s,1),\ o_q(s,1) > \\ < V_q(s,2),\ a_q(s,2),\ o_q(s,2) > \\ \cdots\ \cdots \\ < V_q(s,k),\ a_q(s,k),\ o_q(s,k) >. \end{array}$$

Iteratively, every server s checks the expected rewards obtained by all the actions at each value iteration step, and the k distinct origins which provide the k greatest rewards are recorded in the top-k list. According to (2), (3), and (4), the traditional value iteration can only return the most optimal origin obtained by function max. However, the new value iteration generating the multiple access policy is an extension of the traditional value iteration, which returns the top-k most relevant origins.

Given a query q, the $origin$s reserved in the multiple access policy of our specific MDP are the optimal servers where q is eventually answered. It should be noted that when the MDP policy is computed in our MDP model, we can directly identify the most relevant servers with respect to a query proposed at any server. Therefore, the query can be directly sent to the most relevant servers rather than be routed to them.

3.4 Queries with Multiple Terms

It is clear that setting up the MDP value iteration for every possible query is impossible because of the high computation cost. However, the number of distinct single query terms is limited, especially when we focus on the frequently used terms. Thus, it is feasible to set up the MDP for all single query terms. A query with multiple terms can be answered according to the MDP policies of the query terms contained in the query. From now on, the MDP with top-k policy is constructed according to a single term t instead of a query q. The local immediate reward $r_t(s, a = s)$ is defined as the goodness score of s with respect to query term t.

Suppose query $q =< t_1, t_2 >$, which has two query terms, is submitted to server s. $L_{t1}(s)$ and $L_{t2}(s)$ are the two lists of top-k optimal policies at server s with respect to query term t_1 and t_2 respectively. The total relevance of a server s' to query q is the sum of the relevance scores of all query terms in query q. Lists $L_{t1}(s)$ and $L_{t2}(s)$ are merged into a new list $L_q(s)$ sorted by the adjusted server scores. That is, if s' appears in both $L_{t1}(s)$ and $L_{t2}(s)$, the adjusted relevance score of s' in $L_q(s)$ is the sum of its two expected rewards in $L_{t1}(s)$ and $L_{t2}(s)$. Finally, the new top-k servers are returned. Queries with more than two query terms can be processed in the same manner.

If k, the length of single term policy lists, is large enough, the precision of the query result obtained by this method is still ensured, since analyses [9] demonstrated that the average number of terms in a query is about 2 on the Internet. For example, suppose server s' does not appear in both $L_{t1}(s)$ and $L_{t2}(s)$. Its relevance score using direct merging is 0. If k is appropriately large, the relevance scores of query terms t_1 and t_2 of server s' should be quite small. The possibility of obtaining a large relevance value of query q at server s' is slim, since the relevance value of query q is the sum of two quite small relevance scores of t_1 and t_2.

3.5 Components of the MDP Server

In the value iteration, we iteratively compute $V_{q,i}(s)$ at iteration step i for every server s. Then, every server should act concurrently: every server is in the same iteration step i and has the same number of iteration steps. The concurrency cannot be ensured since there is no centralized control in a flat system where every server is autonomous. Fortunately, there is a variation of value iteration that does not require this assumption. The variant is called *asynchronous value iteration* [11]. In our implementation, we realize the asynchronous value iteration by propagating the reward messages, which are also called MDP messages, in the server network. The algorithm will be explained in detail in Section 4.

Since the server in our system has to fulfill both the query processing task and the asynchronous value iteration task, the search engine will have more components than a traditional one. A search server with MDP utilities will be called

an MDP server from now on. Generally, an MDP server consists of a traditional index server, an MDP calculator, a communicator, and a query retriever.

The index server has two major parts: an index searcher and a local document collection. Given a query, the index server is in charge of retrieving the relevant documents from the local document collection. The query retriever decides whether the query should be answered locally (i.e., answered by the index server) or be directed to the other servers according to the MDP policies. The MDP policies are obtained by the MDP calculator which picks up the MDP value iteration messages from the MDP message queue periodically. All communication tasks are fulfilled by the communicator. It separates the different incoming information (e.g., an MDP value iteration message or a query to be answered) and then passes the information to the corresponding components (e.g., the MDP message queue or the query retriever, respectively). Similarly, the outgoing information is also delivered by the communicator.

4 Algorithm and Complexity

4.1 Passive Asynchronous Value Iteration

In a centralized value iteration, the servers are supposed to be synchronized in the sense that the number of of iteration steps that each server has executed is the same and that they are all executing the same iteration step. In a distributed environment, there is no central control to synchronize the value iteration process in each server; every server only delivers MDP information to its neighbors in the MDP policy computation, and the MDP value iteration has to be carried out in an asynchronous manner. As a consequence of the asynchronous process, a server would not know the next iteration step of its neighbors or if the value iteration had terminated. Since the MDP policy is set up for every keyword, the server has to check the optimal values for every keyword with all the neighbors periodically. It is clear that most of the checks are redundant.

To avoid this endless verification process, we adopt a passive strategy to perform the verification. Instead of inquiring into the neighbors, the server just waits for the arrival of the new V values coming from its neighbors. The expected rewards $V_{t,i}(s)$ for term t at every server s increase monotonically during the value iteration. Moreover, during the value iteration, the current $V_{t,i}(s)$ at s can only be replaced by the new values appearing in its neighbors' policy lists. The current $V_{t,i}(s)$ at s may remain unchanged until its neighbors inform it that the new V values have occurred in their policy lists by sending messages to s. These new V values might potentially replace $V_{t,i}(s)$ with a greater $V_{t,i+1}(s)$, where $V_{t,i+1}(s)$ is the expected reward at the next iteration step at s.

Based on the above discussion, the server can set up the multiple access MDP policies for query terms. Each server maintains a message queue. A message in the message queue has the following form:

$$(t, n, o, V),$$

where $t = term,\ n = neighbor,\ o = origin$, and $V = expected\ reward$. A message with term t may be used to update an item in the multiple access policy list for t.

The MDP message is processed at server s as follows. Picking up an MDP message (t, n, o, V) with term t in the message queue, server s converts it to a potential policy item $< V_p = V, a_p = n, o_p = o >$ for term t at s. Then server s verifies whether the current multiple access policy list for term t at s might be changed by this potential policy item $< V_p, a_p, o_p >$ under the condition that the multiple access policy for t should contain at most k items with k distinct origins.

If the multiple access policy list is updated, it causes more new messages to be sent to the neighbors. For example, if an item in the multiple access policy for t at s is replaced by the potential policy item $< V_p, a_p, o_p >$, a new message $(t, s, o_p, \lambda \cdot V_p)$ is sent to its neighbors except for a_p, which propagates this V_p to server s.

The basic steps of the asynchronous value iteration algorithm returning top-k multiple access policies for query terms are depicted in Figure 1.

Suppose we want to create the multiple access policy for a query term t. A request may be submitted to any server s in the system by sending to s a message $(t, NULL, NULL, 0)$ (of course, we can also send this message to some or all of the servers in the system). After receiving the message and checking its own MDP policy database, s may notice that t is a new term in the policy database. Then, it creates a new policy list for t and inserts into the created policy list the expected reward, which is the local immediate reward $r_t(s, s)$. A message $(t, s, s, \lambda r_t(s, s))$ with discounted reward $r_t(s, s)$ is broadcast to the neighbors of s.

We can notice that the asynchronous MDP value iteration for term t begins with the message propagation and eventually ends with the disappearance of messages about t at every server's MDP message queue, although servers in the system cannot directly perceive the termination of the value iteration.

ε in the algorithm depicted in Figure 1 is a small number provided by the user. The purpose of the small number ε is to reduce the value iteration steps as long as the result is satisfactory at the ε precision level.

The computation complexity of the asynchronous value iteration mostly depends on the number of messages generated in the value iteration as well as on the birth order

Proc MdpCheck(server s, MDP message $m = (t, n, o, V)$)

convert m to the potential policy item $< V_p = V, a_p = n, o_p = o >$

if there is no policy list for t at s do

create the policy list for t at s

compute $r_t(s, s) = G_t$

insert $< r_t(s, s), NULL, s >$ to policy list

send message $(t, s, s, \lambda r_t(s, s))$ to all the neighbors

end if

if o_p appears in the origin of an item $i = (V_i, a_i, o_i)$ in the policy list (i.e., $o_p = o_i$) do

if $V_p <= V_i + \varepsilon$ then return

remove $i = (V_i, a_i, o_i)$

insert $< V_p, a_p, o_p >$ to policy list

send message $(t, s, o_p, \lambda V_p)$ to the neighbors except for a_p

return

end if

if *the number of items in policy list* $< k$ do

insert $< V_p, a_p, o_p >$ to policy list

send message $(t, s, o_p, \lambda V_p)$ to the neighbors except for a_p

return

end if

select item $i = (V_i, a_i, o_i)$, the item with the smallest V value in the policy list

if $V_p <= V_i + \varepsilon$ then return

remove $i = (V_i, a_i, o_i)$

insert $< V_p, a_p, o_p >$ to policy list

send message $(t, s, o_p, \lambda V_p)$ to the neighbors except for a_p

return

End Proc

Figure 1. Asynchronous Value Iteration Returning Top-k Multiple Access Policies

of these messages. Although it is difficult to determine the number of value iteration steps at each server and the message propagating sequence to obtain the exact number of messages generated by the value iteration, we can still provide a very loose upper bound for the number of messages propagated.

Property 1
Given a term t, the upper bound of the total number of messages induced by the asynchronous value iteration is $2NM^2$, where N is the total number of servers in the system and M is the total number of connections between the servers.

Suppose $S = \{s_1, s_2, ..., s_N\}$ is the set of servers and $D = \{d_1, d_2, ..., d_N\}$ is the set of corresponding connection degrees of all the servers. The N servers are the reward providers. For the MDP messages, we have the following fact:

$$\begin{aligned} & the\ total\ number\ of\ messages \\ = \ & the\ total\ number\ of\ messages\ received\ by\ servers \\ = \ & the\ total\ number\ of\ messages\ sent\ by\ servers. \end{aligned}$$

Assume we observe a server $s_i \in S$ and check the maximum number of messages sent by s_i with respect to the reward provided by server s_j.

No matter how many paths exist between s_i and s_j in the server network, the path length of any of them won't exceed M if only paths with no cycles are taken into account. The paths with cycles will be short-circuited by those without during the value iteration because of the discount parameter λ. As such, only paths without cycles are effective routes for MDP messages.

Messages with rewards provided by s_j may be propagated to s_i along the paths between s_i and s_j. Since the maximum path length is M, at most M variations (i.e., values) of the discounted rewards originating from s_j can arrive at s_i. In the worst case, each of them induces a reward replacement at s_i and causes s_i to send d_i messages to its neighbors. The total number of these messages is $M \cdot d_i$.

Similarly, in the worst case, each of the other servers causes s_i to send $M \cdot d_i$ messages to its neighbors. Thus, the maximum number of messages sent by s_i is $N \cdot M \cdot d_i$. Therefore, the maximum number of messages sent by all the servers is:

$$\begin{aligned} & N \cdot M \cdot d_1 + N \cdot M \cdot d_2 + ... + N \cdot M \cdot d_N \\ = \ & N \cdot M \cdot (d_1 + d_2 + ... + d_N) \\ = \ & N \cdot M \cdot 2M \\ = \ & 2NM^2. \end{aligned}$$

Actually, Property 1 is an upper bound that is unlikely to be reached since not every server can appear in all the other servers' top-k reward lists, and the worst case of the reward replacement may not happen because the propagation of many temporary rewards is stopped by the optimal rewards during the value iteration.

4.2 Optimization

In terms of the server network (graph), suppose $SB(s_i)$ is the set of breadth-first spanning trees with root s_i (i.e., each element in $SB(s_i)$ is a breadth-first spanning tree starting from s_i).

Property 2
Given a term t, if the reward originating from server s_i can appear in some other servers' policy lists when the MDP policy is computed, there exists a tree $ply\_tree(s_i)$ with root s_i such that the propagated reward originating from server s_i appears in the policy list of every node in $ply\_tree(s_i)$ and $ply\_tree(s_i)$ is a subtree of one of the elements in $SB(s_i)$.

In other words, the effective and optimal propagation paths for the reward originating from s_i must be a breadth-first spanning tree rooted at s_i. In the breadth-first spanning tree with root s_i, the length of the path between s_i and another server s_j is equal to the distance between s_i and s_j. Thus, the reward originating from s_i and propagated along a path in the breadth-first spanning tree always receives the least discount.

That is, if the reward originating from s_i is propagated along the paths in a breadth-first spanning tree rooted at s_i, it is least discounted when it reaches s_j. If this reward can appear in the policy list at s_j, then it must appear in the policy list of the parent of s_j and can be further propagated to the children of s_j in the breadth-first spanning tree rooted at s_i. If this reward cannot appear in the policy list at s_j, the further propagation of this reward originating from s_i stops since the rewards in the policy list at s_j, which can also be further propagated, have greater values than this reward. Thus, once the MDP policy is computed, the servers whose policy lists contain the reward originating from s_i form a subtree of the breadth-first spanning tree rooted at s_i.

Therefore, in the MDP value iteration, for the reward originating from s_i, it is sufficient to just propagate the reward along a breadth-first spanning tree rooted at s_i since this propagation is already effective and optimal.

If the rewards are propagated along the breadth-first spanning trees with respect to their corresponding origins, we need to obtain a breadth-first spanning tree for each server. We should also let every server know the children of the trees rooted at other servers. In other words, with respect to an origin, each server knows which servers among its neighbors are in the optimal propagation direction.

We assume that the administrator of the distributed search system maintains some concise information about every participant server. If we assume every server's neighborhood is also collected and kept by the administrator (i.e., the administrator has the server network graph), then the breadth-first spanning tree rooted at each server can be readily computed by a simple breadth-first traversal of the graph, which begins with the server. Then, for each breadth-first spanning tree, the administrator sends every server the corresponding children of this tree, which will be kept and used by every server in the message propagation. Since each server only has to store the children's addresses of every server's (origin's) corresponding tree, the additional storage requirement is very small.

Our MDP iteration algorithm is then slightly updated. When a message arrives at the server and causes the renewal of the policy list, instead of sending new messages to every neighbor (except for the neighbor who sends this message), the server only sends new messages to its children according to the tree rooted at the $origin$ of the received message.

In the revised iteration algorithm, the messages are always sent in the optimal directions which lead to the minimum discount. A reward originating from server s_i can be sent to every other server at most once. For a system consisting of N servers, the total number of messages induced by the reward originating from s_i is at most N. Since there are N origins (servers), given a term, the total number of messages sent by the revised iteration algorithm is at most $N \cdot N = N^2$ (Property 3).

Property 3
Given a term t, if the reward originating from any server is always sent in the optimal direction, the upper bound of the total number of messages induced in the revised asynchronous value iteration is N^2.

In our MDP model, the workload of reward propagation is distributed to all the servers in the system. In a balanced flat server network, on average, every server processes the same amount of work. From the viewpoint of the entire system, the upper bound of the message propagation is $N^2/N = N$. Moreover, since the selection of the optimal rewards is conducted during the reward propagation, many small rewards are discarded and this also leads to fewer messages being sent by the servers.

5 Experimental Results

We construct a pseudo-distributed environment to test the effectiveness of our MDP search model. The experi-

ments, which are based on standard retrieval collections, reveal that our MDP model can offer satisfactory results compared to that of a centralized search system.

5.1 Experimental Configuration

TREC[3] Volume 4 and Volume 5 collections, which are stored in TREC Disk 4 and Disk 5 respectively, are adopted in the experiments. The total size of the test data is about $2.1G$. There are approximately $520,000$ documents.

Queries adopted in the experiments are TREC topics from 301 to 350. Each query consists of three parts (i.e., title, description, and narrative) which can be used to form short, moderate, or long queries. Only the titles of TREC topics (queries) are used in our experiments since we are dealing with the Internet queries which are normally short. After the removal of common words and stemming, the average query length of queries 301-350 is 2.48. The longest query contains four words while the shortest one contains only one word.

The total number of pseudo-servers in the system is 128. All the pseudo-servers with indices from 1 to 128 randomly form a flat server network. On average, each server is connected to 5.22 other servers. The maximum connection degree of a single server is 9 and the minimum connection degree is 2. The average distance between two servers in the system is 3.11. The longest distance between two servers is 5. It is apparent that the shortest distance between two servers equals 1.

In order to construct the pseudo-distributed search system, the TREC collections are divided and dispatched to each pseudo-server, that is, each server maintains its own small document collection which is a subset of the TREC collections.

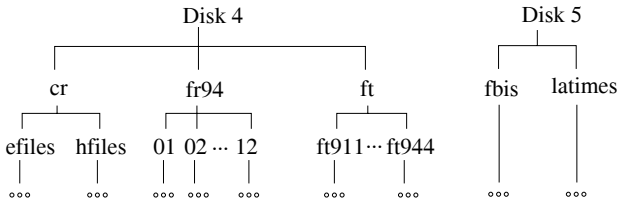
TREC Disk 4 and Disk 5 contain a list of directories, each directory contains a number of files, and each file contains an undetermined number of *retrieval documents*. Figure 2 depicts the directory structure of TREC Disk 4 and Disk 5. In this figure, it can be seen that TREC V4 and V5 contain five macro-domains denoted by their corresponding directory names (i.e., cr, $fr94$, ft, $fbis$, and $latimes$).

For each macro-domain (e.g., $fbis$), we divide the documents in this domain into 128 parts. If these parts are numbered from 1 to 128, given a document in this domain, part i's probability of obtaining this document is P_i, $\sum_{i=1}^{128} P_i = 1$. In our experiments, a skew distribution is adopted. P_i is depicted in Figure 3.

After every domain is distributed to its respective 128 parts according to the skew distribution, for each domain, we randomly map its own 128 parts to 128 pseudo servers. In other words, each server's local collection contains five parts coming from five different domains. For example, in our configuration, the local collection of the pseudo-server with server number one consists of part 82 of $fbis$, part 95 of $latimes$, part 16 of cr, part 44 of $fr94$ and part 120 of ft.

In order to evaluate the effectiveness of our distributed system, precision and recall [1] are used. Precision is defined as the ratio between the number of retrieved relevant documents and the number of retrieved documents. Generally, the precision of top documents is our major concern. Recall is defined as the ratio between the number of retrieved relevant documents and the total number of relevant documents in the entire document collection. Given a value of recall between 0 and 1, we often want to check the corresponding precision value which may be obtained by interpolation. Thus, figures such as Figure 4 can be depicted. If a curve in the figure is closer to point $(1, 1)$, the corresponding performance will be better.

Furthermore, suppose there is an imaginary server which indexes all the documents in every server in the distributed system (i.e., it contains a global document collection). If the search operation conducted by the imaginary server is referred to as *central search*, then a convenient method of checking the effectiveness of our distributed search is to compare the distributed search results with the corresponding central search results. Therefore, we may define the



Note: ◦◦◦ stands for files containing the retrieval documents

Figure 2. Directory Structure of TREC V4 and V5

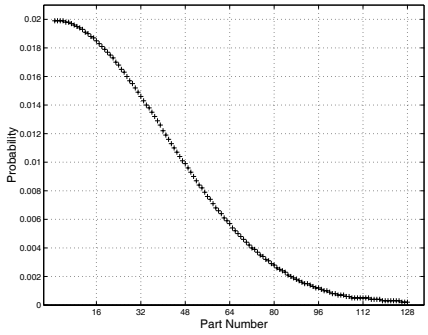


Figure 3. Probabilities of Division Parts

[3] *http://trec.nist.gov/*

precision ratio as follows:

$$Precision\ Ratio\ =\ \frac{Precision\ of\ Distributed\ Search}{Precision\ of\ Central\ Search}\ .$$

Given a query, only the most optimal servers are accessed in the distributed search systems. At the server ranking stage, we adopt the direct merging method for the MDP policy lists since the queries are short and well formed. Once every selected server is visited, each server returns a list of documents sorted by their local TFIDF scores to the server where the query is proposed. We need to merge these document lists into one final list which is presented to the terminal user. To merge these document lists, suppose document d is retrieved from server s. We calculate the weight w_d of d in the final list as follows:

$$w_d\ = ltfidf_d \times\ log(1 + V_s),$$

where $ltfidf_d$ is the local TFIDF weight of d and V_s is the MDP reward of server s in the merged policy list at the server where the query is proposed. Documents are sorted according to their w_d weights in the final document list returned to the user.

5.2 Results

The experimental results unveil the general performance of adopting the MDP method in the distributed search system, which turns out to be effective.

Visiting more servers potentially reduces the possibility of losing relevant documents, whereas visiting few most-relevant servers improves the efficiency. Trade-off should be made according to the user's requirements.

From Figure 4, we may notice that the difference between the central search and the distributed search with cast number 64 (half of the servers in the system) is small, especially when we focus on the top documents (i.e., the corresponding precision values at low recall values in the figure). This can be further confirmed by Figure 6. The precision ratio values for top documents are almost equal to 1 if the cast number is 64. Even if the cast number is only 16, the precision ratio values for top documents are still larger than 0.7. In other words, we visit only an eighth of the servers, but the effectiveness is about 70% of the central search in terms of the top documents which are often browsed by the user.

Visiting more servers always improves the precision for top documents (Figure 5), but the enhancement gradually becomes slim since the most relevant servers which are identified by our MDP policies are visited first, and the rest of the servers in the system are not very qualified.

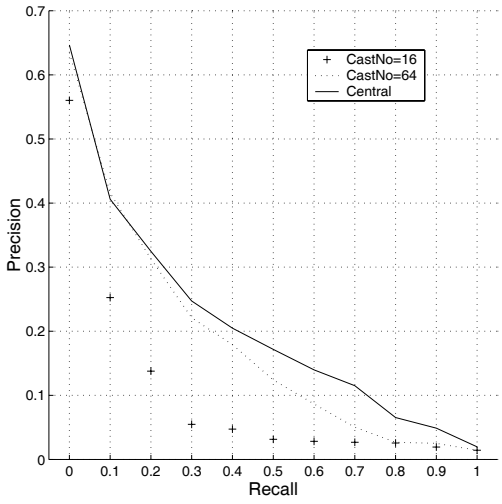


Figure 4. The Precision-Recall of the Cast Number

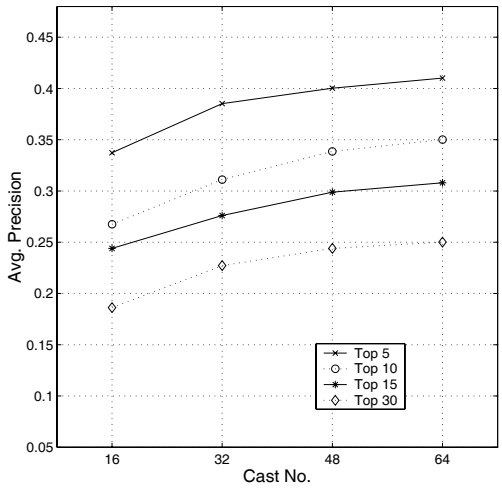


Figure 5. The Cast Number and the Precision of Top Documents

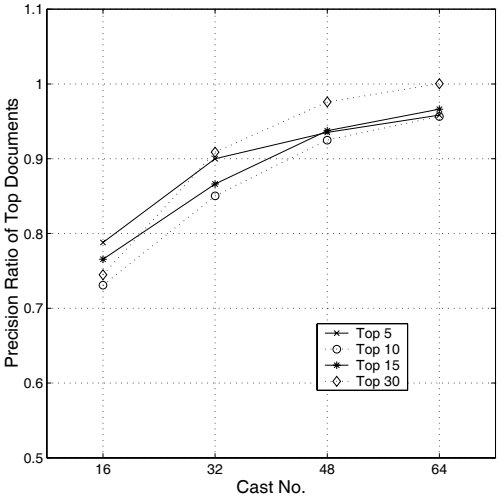


Figure 6. The Cast Number and the Precision Ratio of Top Documents

6 Conclusion

Distributed search systems based on a flat architecture are flexible and scalable for the web environment since every server is autonomous and communicates only with its neighbors. However, it is quite difficult for a server to locate the most relevant servers in the system if a query is proposed to it.

We construct MDPs in a flat distributed search system to facilitate server selection as well as query execution. The MDP policies are computed via MDP value iteration, which has to be adapted in order to maintain the autonomy of each server in our distributed search system. The asynchronous value iteration algorithm based on the propagation of MDP messages is designed for our specific MDP model. This algorithm is carried out autonomously by every participating server in the system. The upper bound of the MDP message propagation is also given in this paper.

We show that the MDP model can successfully address the server selection problem since the MDP policies which are calculated in advance can guide the query to the most relevant servers. The effectiveness of our MDP model is verified by the experiments. It is shown that the distributed search can achieve about the same qualified results as the results returned by the centralized search.

7 Acknowledgments

The research was supported by Research Grant Council, Hong Kong SAR, China under grant numbers HKUST 6154/98E and AoE/E-01/99.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, Essex, England, 1999.

[2] M. Q. W. Baldonado, C.-C. K. Chang, L. Gravano, and A. Paepcke. The Stanford digital library metadata architecture. *International Journal on Digital Libraries*, 1(2):108–121, 1997.

[3] B. Cahoon and K. S. McKinley. Performance evaluation of a distributed architecture for information retrieval. In *SIGIR*, pages 110–118, 1996.

[4] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR*, pages 21–28, 1995.

[5] P. Francis. A global, self-configuring information discovery infrastructure. In *Proceedings of the SIGIR'96 Workshop on Networked Information Retrieval*, Switzerland, 1996.

[6] J. C. French, A. L. Powell, C. L. Viles, T. Emmitt, and K. J. Prey. Evaluating database selection techniques: A testbed and experiment. In *SIGIR*, pages 121–129, 1998.

[7] L. Gong. JXTA: a network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[8] L. Gravano, H. Garcia-Molina, and A. Tomasic. GLOSS: Text-source discovery over the Internet. *ACM Transactions on Database Systems (TODS)*, 24(2):229–264, 1999.

[9] B. J. Jansen, A. Spink, J. Bateman, and T. Saracevic. Real life information retrieval: A study of user queries on the Web. *SIGIR Forum*, 32(1):5–17, 1998.

[10] A. L. Powell, J. C. French, J. P. Callan, and M. Connell. The impact of database selection on distributed searching. In *SIGIR*, pages 232–239, 2000.

[11] M. L. Puterman. *Markov decision processes : discrete stochastic dynamic programming*. John Wiley & Sons, New York, 1994.

[12] Y. Shen and D. L. Lee. A meta-search method reinforced by descriptors of clusters. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE01)*, pages 125–132, Kyoto, Japan, 2001.

[13] C. L. Viles and J. C. French. Dissemination of collection wide information in a distributed information retrieval system. In *SIGIR*, pages 12–20, 1995.

[14] C. T. Yu, W. Meng, K.-L. Liu, W. Wu, and N. Rishe. Efficient and effective metasearch for a large number of text databases. In *CIKM*, pages 217–224, 1999.

[15] B. Yuwono and D. L. Lee. Server ranking for distributed text retrieval systems on the Internet. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 41–50, Melbourne, Australia, 1997.