Enhancing Search Performance on Gnutella-Like P2P Systems

Yingwu Zhu and Yiming Hu, *Senior Member*, *IEEE*

Abstract—The big challenges facing the search techniques on Gnutella-like peer-to-peer networks are search efficiency and quality of search results. In this paper, leveraging information retrieval (IR) algorithms such as Vector Space Model (VSM) and relevance ranking algorithms, we present GES (**G**nutella with **E**fficient **S**earch) to improve search performance. The key idea is that GES uses a distributed topology adaptation algorithm to organize semantically relevant nodes into same semantic groups by using the notion of node vector. Given a query, GES employs an efficient search protocol to direct the query to the most relevant semantic groups for answers, thereby achieving high recall with probing only a small fraction of nodes. To the best of our knowledge, GES is the first to identify node vector size as an important role in impacting search performance and to show that the node vector size offers a good trade-off between search performance and bandwidth cost. Moreover, GES adopts automatic query expansion and local data clustering to improve search performance. We show that GES is efficient and even outperforms the centralized node clustering system SETS. For example, in the scenario where node capacity is heterogeneous, GES can achieve 73 percent recall when probing only 20 percent nodes, outperforming SETS by about 18 percent.

Index Terms—Peer-to-peer, topology adaptation, biased walk, semantic group, node vector, recall, information retrieval.

1 Introduction

In the past few years, peer-to-peer (P2P) networks such as Gnutella have become some of the fastest growing and most popular Internet applications. They have demonstrated the significance of distributed information sharing by spreading storage, information, and computation cost among nodes. To facilitate sharing of information, P2P networks should provide an *efficient* and *versatile* search functionality—the search mechanism should not only be efficient in terms of resource consumption (e.g., bandwidth), but also support *rich* (or *complex*) queries such as keyword search and content-based full-text search.

Distributed Hash Tables (DHTs) [1], [2], [3], [4] and unstructured P2P networks such as Gnutella provide different data location mechanisms. DHTs are adept at *exact-match* lookups: given a key, they can locate the corresponding document with $O(\log N)$ overlay hops in a network of N nodes. However, extending exact-match lookups to support complex queries on DHTs is nontrivial [5], [6], [7], [8]. On the other hand, unstructured P2P networks, like Gnutella, organize nodes into a random graph and rely on flooding queries on the graph to retrieve relevant documents. Each visited node evaluates the query locally on its own content and thus arbitrarily complex queries can be easily supported on Gnutella-like P2P networks. But, the main drawback is search inefficiency—either a large fraction of nodes have to be probed or some relevant documents have to be missed.

A number of search solutions [9], [10], [11], [12], [13] have been proposed to improve search performance on Gnutella-like systems. However, with few exceptions [13], most of them ignore information retrieval (IR) algorithms such as Vector Space Model (VSM) and relevance ranking algorithms, and thus may not be able to provide high quality of search results (e.g., *high recall*). Worse yet, a query containing popular terms may return a superfluous number of documents beyond a user's capability to deal with. GES (**G**nutella with **E**fficient **S**earch), proposed here, combines techniques from IR and P2P computing to enhance search performance in terms of *search efficiency* and *quality of search results*. For example, with the IR algorithms such as VSM and relevance ranking algorithms, GES may return the documents with the highest relevance scores which are usually considered relevant to a query, thereby avoiding forcing users to deal with a superfluous number of documents. Moreover, exploiting IR algorithms such as automatic query expansion [14] helps improve recall (which measures completeness of search results) and *precision* (which measures purity of search results).

1.1 Overview of GES

The design philosophy underlying GES is that GES aims to improve search performance while retaining the simple, robust, and fully decentralized nature of Gnutella. GES is based on the intuition: if nodes are semantically relevant (i.e., they have similar content), it is likely that they are relevant to the same queries. To determine whether two nodes are semantically relevant or not, GES introduces a notion of *node vector*, a compact summary derived from a node's documents using VSM. Node vectors are used to calculate node relevance. If the relevance score of two nodes' node vectors is high (e.g., higher than a certain

- *Y. Zhu is with the Department of Computer Science and Software Engineering, Seattle University, Seattle, WA 98122-1090. E-mail: zhuy@seattleu.edu.*
- *Y. Hu is with the Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221-0030. E-mail: yhu@ececs.uc.edu.*

Manuscript received 29 Jan. 2005; revised 1 Aug. 2005; accepted 29 Oct. 2005; published online 25 Oct. 2006.
Recommended for acceptance by D. Bader.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0049-0105.

relevance threshold), then the two nodes are semantically relevant; otherwise, they are irrelevant.

The main components of GES are the *distributed topology adaptation algorithm* and the *search protocol*. The topology adaptation algorithm is performed periodically at each node to restructure the overlay such that semantically relevant nodes are organized into same *semantic groups* through *semantic links*. Each node also connects to some *irrelevant* nodes through *random links* by which GES can efficiently discover different semantic groups. The search protocol is a mix of *biased walks* and *flooding*. Given a query, GES first uses biased walks through random links to locate a relevant semantic group for the query, and then floods the query through semantic links within the semantic group to retrieve relevant documents. The search will continue this process until sufficient answers are found.

By exploiting IR algorithms, can GES improve search performance in terms of efficiency and quality of search results? The answer mainly depends on *quality of semantic groups* and *efficiency of the search protocol*.

The quality of semantic groups measures *goodness* of a semantic group (i.e., nodes within a group are truly semantically relevant), and is ultimately determined by node vectors, which are used to calculate node relevance. To the best of our knowledge, GES is the *first* to identify *node vector size*, the number of terms in a node vector, as an important role in impacting node relevance characterization and ultimately the quality of semantic groups.

On the assumption that nodes within a semantic group tend to be relevant to the same queries, the efficiency of the search protocol depends on the efficiency of relevant semantic group discovery for queries. Relying on *one-hop node vector replication*, GES can quickly route the query to the relevant semantic group by biased walks through random links. Moreover, the capacity-aware mechanism of the search protocol can exploit node capacity heterogeneity to speed up the discovery of the relevant semantic group, thereby improving search performance.

GES does not assume the documents on a node are restricted to one single topic/area. A node's documents could be diverse in topics. The diversity of documents may complicate the task of characterizing node relevance and affect the quality of semantic groups. We thus explore *local data clustering* to improve the quality of semantic groups, with the aid of *virtual nodes* [1]. Moreover, we introduce the IR technique, automatic query expansion, to improve the quality of search results in terms of recall and precision.

1.2 Structure of the Rest of the Paper

The remainder of the paper is structured as follows: Section 2 gives an overview of related work. Section 3 provides necessary background on VSM. In Section 4, we describe the design of GES. In Section 5, we present our evaluation metrics, data, and simulation methodology. We provide our experimental results in Section 6. We finally conclude the paper in Section 7.

2 Related Work

A number of P2P keyword search systems [5], [6], [15] have been built on top of DHTs. All of them are based on global indexing where each node is responsible for the inverted list of some keywords (or terms). Reynolds et al. [6] proposed three techniques for multiple keyword search, including *Bloom filters*, *caches*, and *incremental results*, to minimize the amount of bandwidth consumed by intersection of inverted lists and reduce end-user perceived latency. Li et al. [5] focused their study on the feasibility analysis for multiple keyword search based on the resource constraints and search workload, and showed that the combination of optimizations (e.g., clustering and gap compression) and compromises on quality of search could make P2P-based keyword search within feasibility range. eSearch [15] exploits semantic information produced by IR algorithms to selectively replicate complete lists of terms for documents based on top terms, thereby avoiding intersection of inverted lists in multiple keyword search.

Recently, some content-based full-text search systems [7], [8] based on DHTs have been proposed. pSearch [7] distributes document indices to peer nodes based on document semantics generated by Latent Semantic Indexing (LSI). The indexes of semantically related documents are likely colocated in the overlay, thereby achieving good search performance at low processing cost. In a similar vein, Zhu et al. [8] proposed a content-based search system by taking advantage of VSM and Locality Sensitive Hashing (LSH). The indexes of semantically close documents are stored on the same peer nodes with high probability (i.e., nearly 100 percent), and a query only needs to search a small number of nodes (e.g., 20) for answers. Tang et al. [16] proposed an eLSI algorithm to improve efficiency of LSI, by using techniques such as document clustering plus term selection or random projection. They showed that eLSI makes pSearch more efficient while retaining retrieval quality of LSI.

However, node churn in P2P networks complicates the task of the DHT-based search solutions. To deal with node churn, AESOP [17] imposes a hierarchy structure onto DHTs: it partitions a DHT into DHT-structured clusters each of which has at least one *altruistic* peer that undertakes more responsibilities, and improves routing efficiency even in the face of high churn with the aid of altruistic peers. Gupta et al. [18] proposed two novel P2P lookup algorithms, with routing performance of one and two hops, respectively. Their analytic results show that the proposed solutions consume reasonable bandwidth even in the presence of high churn. Bamboo [19] addresses the issue of how to handle churn in DHTs. It identifies and explores three factors that affect DHT performance under churn, including reactive versus periodic failure recovery, message timeout calculation, and proximity neighbor selection (PNS).

Improvements to Gnutella's flooding mechanism have been studied along three dimensions: *random walks*, *guided search*, and *group-based search*. Lv et al. [9] proposed random walks in place of flooding to improve scalability. Random walk is essentially a blind search in that at each step a query is forwarded to a randomly chosen node without considering any hint of how likely the next node will have answers for the query. To overcome the blindness of random walks, Crespo et al. [11] introduced the notion of "routing indices" to guide a query toward nodes which are more likely to

have the requested documents. This search technique is similar to GES's biased walks which rely on replicated node vectors to direct a query toward most relevant nodes for answers. Systems such as [10], [12], [13], [20] organize nodes into groups to improve search efficiency on Gnutella-like P2P systems. However, with very few exceptions [13], most of them ignore the IR algorithms and, thus, may not be able to provide guarantee on recall. Cohen et al. [10] used guide-rules to organize nodes satisfying some predicates into an associative network. In a similar vein, Sripanidkulchai et al. [12] used interested-based locality to organize nodes into interest-based structure, by which a significant amount of flooding on Gnutella-like systems can be avoided.

The closest work to GES is SETS [13]. SETS is a search system using a topic-driven query routing protocol on a topic-segmented overlay built from Gnutella-like P2P systems. A topic segment in SETS contains nodes with similar content and is similar to a semantic group in GES. The topic-segmented overlay is constructed by performing node clustering at a *single* designated node, and each cluster corresponds to a topic segment (nodes within a topic segment are connected by *local links* while nodes belonging to different topic segments are connected by *long-distance links*). Given a query, SETS first computes R topic segments which are most relevant to the query and then routes the query to these segments for relevant documents. When a node joins the system, it first has to contact the designated node for the information about all the C topic segments and then joins the most relevant segment. Moreover, as nodes join/leave or their document collections change, the designated node has to recompute topic segments to keep them up-to-date and then disseminates them throughout the system. Several important features distinguish GES from SETS. First, GES uses a distributed topology adaptation algorithm to organize nodes into semantic groups while in SETS a single designated node is responsible for clustering nodes into topic segments. Such a centralized structure may suffer from a single point of failure and performance bottleneck. Second, GES's search protocol is capacity-aware, and it can exploit node capacity heterogeneity to significantly improve performance. Third, GES is the first to identify the important role of node vector size in search performance, and to show that an appropriate node vector size is very important in system design. Finally, GES is the first to adopt automatic query expansion and data clustering into Gnutella-like P2P systems to improve search performance.

The work done by Triantafillou et al. [21] is among the first to propose peer clustering to improve search performance. The nodes are organized into a set of clusters and each node joins the clusters corresponding to the document categories it belongs to. Routing efficiency is achieved by forwarding the query to the relevant clusters. The clusters are similar to the semantic groups in GES. However, this work does not study the issues of how to derive document categories, how to form semantic clusters, and quality of search results.

PlanetP [22] uses *Bloom filters* to summarize content on each node and floods the summaries to the entire system. OceanStore [23] proposes a probabilistic location algorithm to improve the location latency of existing DHT's deterministic lookups, if the replica of a requested document exists close to query sources. The approach is based on *attenuated bloom filters*, a lossy distributed index structure constructed on each node. Due to the attenuated Bloom filters maintained along each neighbor link, the algorithm can find the nearby replicas very quickly. GES, instead, uses node vectors to summarize content on each node and relies on replicated node vectors to locate the relevant semantic group for queries. To overcome the limitations of search on both structured and unstructured P2P systems, Loo et al. [24] proposed a hybrid search technique on Gnutella-like systems in which structured P2P search techniques are used to index and locate rare documents and flooding techniques are used to locate highly replicated documents.

Gia [25] combines a number of techniques, including random walks, topology adaptation, replication and flow control, to improve scalability of Gnutella-like P2P networks. GES's topology adaptation partly draws inspiration from Gia's. However, GES's topology adaptation differs from Gia's in that it is mainly used to form semantic groups to improve search performance while Gia uses topology adaptation to improve system scalability.

Recent work [26], [27] explores the issue of information retrieval in the environment of distributed collections. COSCO [26] concentrates on gathering coverage and overlap statistics of collections and uses these statistics at query time to best estimate which set of collections should be searched. Bender et al. [27] proposed an overlap-aware P2P Web search engine which considers mutual overlap among collections. They have showed that the combination of the mutual overlap estimates among collections with existing quality estimation metrics can dramatically decrease the number of collections that have to be contacted in order to achieve a satisfactory level of recall. Viewing semantic groups in GES as collections, these proposed techniques may be applicable to GES in identifying the semantic groups that have to be searched for a query.

3 Background: Vector Space Model (VSM)

In VSM [28], each document/query is represented by a vector of terms. The terms are stemmed words[1] which occur within the document/query. In addition, stop words[2] and highly frequent words are removed from the term vector. Each term in the vector is assigned a weight by a term weighting scheme. Terms with heavy weight are generally deemed to be central to a document. To evaluate whether a document is relevant to a query, VSM measures the relevance between the query vector and the document vector. Typically, for a document D and a query Q (suppose the term vectors of D and Q have been already normalized), the relevance score is computed as:

$$REL(D,Q) = \sum_{t \in D,Q} d_t \cdot q_t, \tag{1}$$

1. Stemmed words are the words that provide the root for other related words. For example, the stemmed word for words *restarted*, *restarts*, and *restarting* is *restart* by removing their suffixes.

2. Stop words are those words that are considered noninformative, like function words *of*, *the*, *a*, etc.

where t is a term appearing in both D and Q, q_t is term t's weight in query Q, and d_t is term t's weight in document D. Documents with high relevance scores are deemed to be relevant to the query.

A number of term weighting schemes have been proposed, among which *tf-idf* is a scheme in which the weight of a term is assigned a high numeric value if the term is frequent in a document but infrequent in other documents. The main drawback of *tf-idf* is that it requires some *global* information (i.e., the document frequency *df*, which represents the number of documents where a term occurs) to compute a term's weight. Obtaining the global information needs to perform information aggregation and dissemination throughout the system, which is not an easy task in the P2P environment. Thus, GES uses a "dampened" *tf* scheme where each term t is assigned a weight in the form of $d_t = 1 + \log f_t$, where f_t is t's term frequency in a document. The "dampened" *tf* scheme has two main advantages: 1) This scheme does not require any global information and 2) it produces high quality document clusters [29].

4 System Design

In this section, we detail the design of GES. We in turn describe node vector, discuss topology adaptation algorithm, present one-hop node vector replication, and discuss search protocol.

4.1 Node Vector

Node vector summarizes a node's content and is used to determine relevance between nodes. A node X's node vector is derived from its documents using VSM as follows. First, each document i is represented by a *temporary* term vector where each term t's weight is represented by its term frequency $f_{t,i}$. Second, all temporary term vectors of X's documents are summed up, and we get a new vector in which each term component t has a weight $f_t = \sum_{i=1}^{n} f_{t,i}$, where n is the number of documents on X. For each term t, we replace its weight f_t with $1 + \log f_t$ by using the "dampened" *tf* scheme. Finally, we normalize the new vector and the normalized vector is X's node vector.

Given two nodes X and Y, their relevance score is computed as:

$$REL(X, Y) = \sum_{t \in X,Y} w_{X,t} \cdot w_{Y,t}, \quad (2)$$

where t is a term appearing in both X and Y, $w_{X,t}$ is term t's weight in X, and $w_{Y,t}$ is term t's weight in Y. If the relevance score is less than a certain relevance threshold, nodes X and Y are deemed to be irrelevant; otherwise, they are deemed to be relevant.

Node vectors are also used to calculate the relevance of a node X and a query Q according to (3), as will be shown later in biased walks during search.

$$REL(X, Q) = \sum_{t \in X,Q} w_{X,t} \cdot w_{Q,t}. \quad (3)$$

4.2 Topology Adaptation Algorithm

The task of topology adaptation is to restructure an initial P2P overlay such that 1) semantically relevant nodes are organized into the same semantic groups through semantic links and 2) each node maintains some random links to facilitate semantic group discovery. The topology adaptation algorithm is fully distributed: Each node periodically performs *neighbor discovery* (Section 4.2.1) and *neighbor adaptation and maintenance* (Section 4.2.2) to adjust its neighbor links, thereby ultimately altering the overlay topology.

4.2.1 Neighbor Discovery

Each node may have two types of neighbors: *random neighbors* (the semantically irrelevant nodes, which are connected by random links) and *semantic neighbors* (the semantically relevant nodes, which are connected by semantic links). To adapt to the churn in node memberships as well as the changes of node's documents (e.g., document addition or removal), each node periodically issues random walk queries to discover new neighbor candidates.

A random walk query message contains the query originator's node vector, a relevance threshold *REL_THRESHOLD*, the maximum number of responses *MAX_RESPONSES*, and TTL (time-to-live). The random walk returns a set of nodes. Actually, each node periodically issues two queries, one requesting nodes whose relevance is lower than *REL_THRESHOLD*, and the other requesting nodes whose relevance is higher than or equal to *REL_THRESHOLD*. Note that the relevance score is computed using (2) (in Section 4.1). The returned nodes are added to the query originator node's two neighbor candidate caches: *random neighbor cache* and *semantic neighbor cache*, according to their relevance scores. Each cache entry consists of *a node's IP address*, *port number*, *node capacity*, *node degree*, *node vector*, and *relevance score*.[3] These two caches are maintained throughout the lifetime of the node. Moreover, each cache has a size constraint and simply uses FIFO as replacement strategy.

4.2.2 Neighbor Adaptation and Maintenance

The task of neighbor adaptation and maintenance includes selecting new neighbors from the neighbor candidate caches and keeping track of the existing neighbors.

Each node (say, X) periodically updates its semantic neighbors by choosing new neighbors from the semantic neighbor cache, as outlined by $adapt\_sem\_neighbors()$ in Fig. 1. MAX_SEM_LINKS is the maximum number of semantic neighbors a node can have. Note that the node attempts to choose the most relevant node as semantic neighbors, while avoiding dropping the already poorly connected semantic neighbors.[4] In addition, each node periodically updates its random neighbors by selecting new neighbors from the random neighbor cache, as outlined by $adapt\_rnd\_neighbors()$. MAX_RND_LINKS is the maximum number of random neighbors a node can have. Note that random neighbor update takes into account node

3. Keeping precomputed relevance scores in cache avoids recomputing.

4. As will be shown in Section 5.3, each node has a minimum degree constraint and a typical value is 3. If a node's degree is less than or equal to the minimum constraint value, this node is identified as a poorly connected node.

```
X.adapt_sem_neighbors() //update semantic neighbors
Y ← choose a node from X's semantic neighbor cache such that it is not dead and not yet X's neighbor and with the highest relevance score
  //a dead node is a node which has already left the system
if SEM_LINKS<MAX_SEM_LINKS then
   X connects to Y
   if Y accepts X then
      Y becomes X's semantic neighbor
      SEM_LINKS++ //increase the number of semantic neighbors
   end if
else
   if Y's relevance score is higher than all of X's semantic neighbors then
      Z ← choose a node from X's semantic neighbors such that it is not poorly connected and with the lowest relevance score
      if Z does not exist then
         Z ← choose a node from X's semantic neighbors such that it has the lowest relevance score
      end if
      X connects to Y
      if Y accepts X then
         X replaces Z with Y //add new neighbor Y and drop existing neighbor Z
      end if
   else
      Z ← choose a node with the lowest relevance score from those X's semantic neighbors whose relevance scores are lower than that of Y
        and which are not poorly connected
      if Z exists then
         X connects to Y
         if Y accepts X then
            X replaces Z with Y //add new neighbor Y and drop existing neighbor Z
         end if
      end if
   end if
end if

X.adapt_rnd_neighbors() //update random neighbors
Y ← choose a node from X's random cache such that it is not dead and not yet X's neighbor and with the highest capacity
if Y.capacity ≤ X.capacity then
   Y ← randomly choose a node from X's random cache such that it is not dead and not yet X's neighbor
end if
if RND_LINKS<MAX_RND_LINKS then
   X connects to Y
   if Y accepts X then
      Y becomes X's random neighbor
      RND_LINKS++ //increase the number of random neighbors
   end if
else
   if Y.capacity is higher than all of X's random neighbors then
      Z ← choose a node from X's random neighbors such that it has the highest degree
      X connects to Y
      if Y accepts X then
         X replaces Z with Y //add new neighbor Y and drop existing neighbor Z
      end if
   else
      Z ← choose a node with the highest degree from those X's random neighbors whose capacities are lower than or equal to that of Y
      if Y.degree < Z.degree then
         X connects to Y
         if Y accepts X then
            X replaces Z with Y //add new neighbor Y and drop existing neighbor Z
         end if
      end if
   end if
end if
```

Fig. 1. Pseudocode for neighbor adaptation.

capacity, hoping high capacity nodes have high degree and low capacity nodes are within short reach of higher capacity nodes. The capacity-aware random neighbor selection, as will be shown later, makes biased walks (in Section 4.4) more efficient in the scenarios where node capacity is heterogeneous.

Each node also continuously keeps track of the node vectors of its existing neighbors. Periodically, the node requests the fresh node vectors from each of its neighbors, and each neighbor responds to the request with its up-to-date node vector. To minimize the bandwidth cost of transferring the node vector, each neighbor can reply with the *update delta* of its node vector. Note that the request messages may be piggybacked onto *keep-alive* messages between the node and its neighbors to reduce the number of messages. Upon receiving the fresh node vectors (or the update deltas), the node checks the relevance scores of its semantic links and random links: if the relevance score of a semantic link drops below $REL\_THRESHOLD$ due to dynamically changing documents in either node, GES simply drops the semantic link and adds the neighbor information into the random neighbor cache. Similarly, if

```
X.biased_walk(Query q) //baised walks
1: if X.is_targetnode(q) then
2:    X.flood(q) //flood the query through X's semantic links
3: else
4:    Y ← X.next_hop(q) //the next hop for biased walks
5:    Y.biased_walk(q)
6: end if

X.flood(Query q) //query flooding
1: for each semantic neighbor S_i do
2:    S_i.flood(q)
3: end for

X.is_targetnode(Query q) //check if X is a semantic group target node for q
1: if X has relevant documents for q then
2:    return true
3: else
4:    return false
5: end if

#define is_supernode(X) (X.capacity < C ? false : true) // C is capacity threshold
X.next_hop(Query q) //choose a next hop for biased walks
1: Y ← choose the random neighbor most relevant to q using Equation 3
2: if is_supernode(X) then
3:    return Y
4: else
5:    Z ← choose the random neighbor with the highest capacity
6:    if is_supernode(Z) then
7:       return Z
8:    else
9:       return Y
10:   end if
11: end if
```

Fig. 2. Pseudocode for search protocol.

the relevance score of a random link rises above $REL\_THRESHOLD$, GES simply drops the random link and adds the neighbor information into the semantic neighbor cache. As a result, the neighbor adaptation and maintenance performed thereafter can adapt to dynamically changing node vectors of each node's existing neighbors.

4.3 One-Hop Node Vector Replication

To assist biased walk decision making, each node maintains the node vectors of *all* of its random neighbors in memory. When a random-link connection is lost, either because the random neighbor fails or due to topology adaptation, the node vector for this random neighbor gets flushed from memory. As discussed in Section 4.2.2, each node keeps track of the node vectors of its existing neighbors. Therefore, each node can keep the replicated node vectors up-to-date and consistent.

4.4 Search Protocol

The search protocol is a mix of biased walks and flooding. Given a query, GES first relies on biased walks (through random links) to locate a relevant semantic group, and then floods the query within this group (through semantic links) to retrieve relevant documents. GES will continue this search process until sufficient answers are found. Note that the query flooding within the semantic group is based on the intuition that semantically associated nodes are likely to be relevant to the same queries.

Fig. 2 shows the pseudocode for the search protocol. Given a query q, node X executes $biased\_walk()$: each document is evaluated using (1) and a relevance score is computed.[5] If the relevance score is high, this document is identified as a relevant document for the query. If any such a relevant document is identified, then the node X is called a *semantic group target node* where the query terminates biased walks and starts flooding. Otherwise, X executes $next\_hop()$ to make a biased walk decision, picking a random neighbor to which X forwards the query. Note that the biased walk decision takes into account node capacity: it favors the high-capacity neighbor, hoping that the high-capacity neighbor can typically provide useful information for the query. The biased walk decision also takes advantage of the replicated node vectors and favors the most relevant random neighbor if there is no high-capacity random neighbor.

During flooding, each visited node evaluates the query against its own documents and floods the query through its own semantic links. The relevant documents found within the semantic group are directly reported to the target node. Note that each query contains a $MAX\_RESPONSES$ parameter. The target node aggregates the relevant documents, reports them directly to the query initiator node which will present highest relevance ranking documents to the user, and decreases $MAX\_RESPONSES$ by the number of relevant documents. If $MAX\_RESPONSES$ becomes less than or equal to zero, the query is simply discarded. Otherwise, the query starts biased walks from the target node again and repeats the above search process until sufficient responses are found. In addition to $MAX\_RESPONSES$, each query is also bound by a TTL parameter. Note that flooding within semantic groups keeps us from exactly keeping track of the TTL. For simplicity, GES decreases the TTL by one at each step *only* during biased walks. Once TTL hits zero, the query message is dropped and no longer forwarded.

During both biased walks and flooding, we use bookkeeping techniques to sidestep redundant paths. Each query is assigned a unique identifier $GUID$ by its initiator node, and each node keeps track of the neighbors to which it has already forwarded the query with the same $GUID$. During biased walks, if a query with the same $GUID$ arrives back at a node (say, X), it is forwarded to a different random neighbor chosen by $next\_hop()$ from those random neighbors to which X has *not* forwarded the query yet. This reduces the probability that a query traverses the same link twice. However, to ensure forward progress, if X has already sent the query to all of its random neighbors, it flushes the bookkeeping state and starts reusing its random neighbors. On the other hand, during flooding, if a query with the same $GUID$ arrives back at the node, the query is simply discarded.

5 Performance Evaluation

In this section, we discuss the design of experiments to evaluate GES. We start by discussing performance metrics and data for our evaluation. We then describe our methodology.

5.1 Performance Metrics

The metrics we used to express the benefits and cost of GES are:

5. We may narrow down the local evaluation scope by clustering a node's documents first.

- *Recall*. It is a main metric used to quantify the quality of search results, and is defined as the number of retrieved relevant documents divided by the number of relevant documents.
- *Precision@r*. It is another metric used to quantify the quality of search results, and is defined as the number of retrieved relevant documents divided by the number r of retrieved documents. We are particularly interested in high-end precision (e.g., precision@15) because a recent study [30] has shown that users only view top 10 search results.
- *Query Processing Cost*. It is defined as the fraction of nodes in the system which are involved in a query processing. A lower query processing cost increases system scalability since system resource consumption is proportional to the number of nodes visited by a query. In addition, query processing cost measures the quality of semantic groups. If the quality of semantic groups is poor, a query may probe many irrelevant nodes, thereby increasing query processing cost.
- *Bandwidth Cost (Kbps)*. It is used to quantify the bandwidth overhead incurred by the topology adaptation algorithm (Section 6.4).

5.2 Data

We used two sets of data in our experiments: *TREC-1,2-AP* [31] and *Reuters* [32].

TREC. The TREC corpus is a standard benchmark widely used in the IR community. *TREC-1,2-AP* contains AP Newswire documents[6] in TREC CDs 1 and 2. We extracted those documents with text and valid author fields from this original document collection. We assumed that each author corresponds to a node and his/her associated documents are stored on the corresponding node. This resulted in 80,008 documents distributed over 1,880 nodes. The mean, 1st-percentile and 99th-percentile of the number of documents per node are 42.5, 1, and 417, respectively. The term vector of a document was derived from the text field using VSM. The terms in each document vector were stemmed. We also used a list of 571 stop words from SMART [33] to remove stop words from each document vector. Each document vector on average has 179 unique terms.

We used a set of 50 queries from TREC-3 ad hoc topics, with query number from 151 to 200. The query vector was derived from the title field using VSM. The terms in each query vector were stemmed and stop words were removed as well. Thus, these 50 queries each has, on average, 3.5 unique terms. Moreover, the 50 queries each comes with a query relevant judgment file which contains a set of already identified relevant documents by TREC-3 ad hoc query assessors. Since we only used 80,008 AP Newswire documents with valid author and text fields, we removed those documents which do not appear in this 80,008 document set from the accompanying relevant judgment files.

Fig. 3 shows that about 50 percent of nodes provide relevant documents for two or more queries (e.g., the maximum is 12 queries). We found that the topics of these queries are very different. We conclude that documents created by an author (and, thus, stored on a node) are not restricted to one single topic/area. Hence, the data used in our evaluation does not assume a node specializes in one single topic, and a very large portion of nodes hold diverse documents indeed (the manual checking of documents also confirmed this).

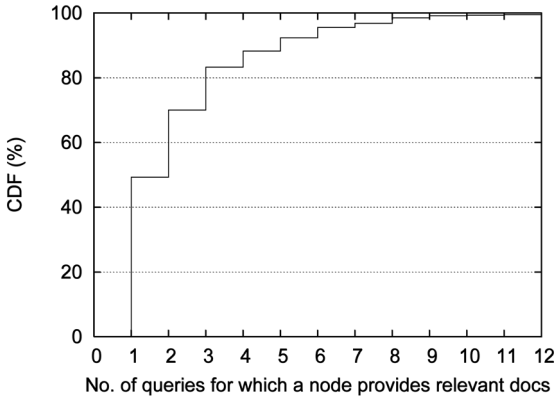


Fig. 3. Cumulative distribution of the number of queries for which a node provides relevant documents.

Reuters. The Reuters data (volume 1) contains news articles each of which has text and author fields. Similarly, we assumed that each author corresponds to a node and his/her associated documents are stored on the corresponding node. This resulted in 109,500 documents distributed over 2,368 nodes. Unlike the TREC data, we generated a set of 100 queries for the Reuters data. Each query was generated by first randomly choosing a document and then picking three terms uniformly from its term vector. The relevant documents for a query are the documents which contains all the terms in the query.

5.3 Methodology

GES simulation starts with a randomly-connected topology, and then uses topology adaptation algorithm to restructure the initial topology. GES's topology adaptation algorithm uses four preconfigured parameters, as shown in Table 1. We set $max\_links = 8$ in the experiments where node capacity is assumed to be uniform. In the experiments where node capacity is heterogeneous, we set max_links to 128. However, there is a constraint on max_links, i.e., $max\_links = min(max\_links, \lfloor \frac{C}{min\_unit} \rfloor)$, where C is a node's capacity and min_unit represents the finest level of granularity into which a node's capacity is split, with a typical value of 4.

For performance comparison, we consider two search systems: *Random* and SETS. Random represents random walks presented in reference [9], and is used as a baseline system. We choose SETS due to two main considerations: 1) Both SETS and GES use the same IR algorithms such as VSM and term weighting scheme. It is fair to compare their performance based on the same IR algorithms, and 2) SETS is a *centralized* node clustering system (as described in Section 2) while GES is a *decentralized* node grouping system. It is interesting to compare their performance with different design philosophies.

6. Associated Press news articles. They are part of the TREC ad hoc document collections.

TABLE 1
The Parameters Used in GES

Parameters	Definition	Value
min_links	the minimum number of neighbors per node	3
max_links	the maximum number of neighbors per node	8-128
α	the fraction of max_links for semantic neighbors	0.5
$node\_rel\_threshold$	node relevance threshold	0.45

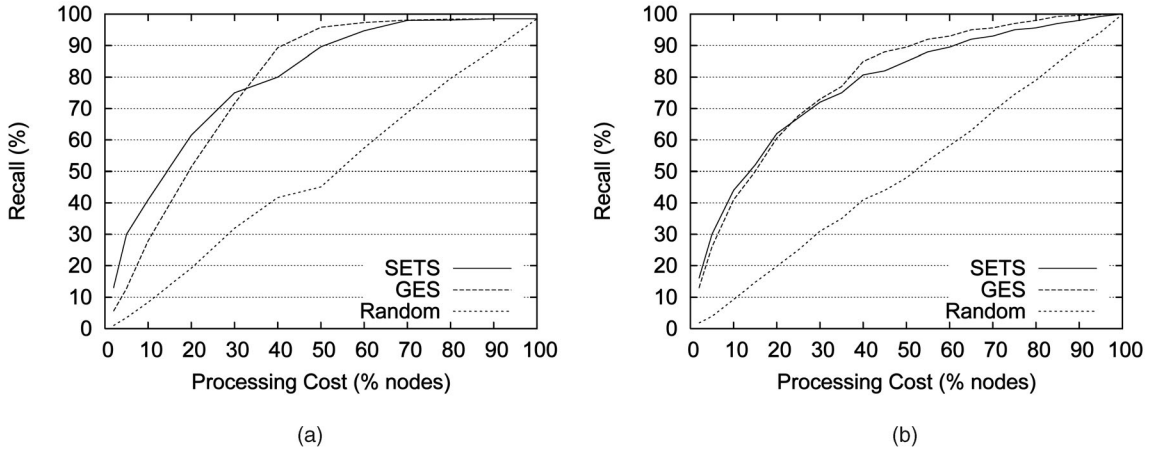


Fig. 4. (a) Recall versus processing cost for TREC. (b) Recall versus processing cost for Reuters.

In Random, there is no topology adaptation, and a random graph is used as the overlay. In SETS, a designated node performs topic segmentation to reconfigure the initial randomly-connected topology. Each node has four long-distance links and four local links. The number of topic segments are 256, which represents the best case for performance compared to 32, 64, and 128 topic segments. Throughout all of our experiments, we choose uniformly random graphs with an average degree of 8. The primary purpose of choosing such uniformly random graphs is to avoid unnecessarily biasing against Random.

In most of our experiments, node capacity is assumed to be uniform unless otherwise specified. The whole node vector on each node is used to compute node relevance in topology adaptation unless otherwise noted. We ran each experiment 10 times with different seeds. The experimental results presented in the next section are the average numbers of the experiments. All tests were run on all the data sets, where the results are similar we present results on one of the data sets due to space limitations.

6 Experimental Results

6.1 Performance Comparison

Fig. 4 shows evaluation results comparing the performance of GES against SETS and Random. Several observations can be drawn from this figure:

1. GES and SETS outperform Random substantially, achieving higher recall at lower query processing cost.
2. Compared to GES, SETS achieves higher recall for the TREC data when probing less than 30 percent of nodes. This is explained by the fact that SETS takes advantage of knowing the global $C(= 256)$ topic segments and, therefore, can quickly and precisely locate the most relevant topic segments to look up relevant documents. GES, instead has to rely on biased walks to locate a semantic group target node, and then floods the query within the semantic group for relevant documents. If the target node is not a right one (which actually does not contain relevant documents, though some of its documents have relevance score high enough to be deemed relevant), some irrelevant nodes are unavoidably probed. The overhead of locating a right target node hurts the performance of GES, especially when probing *only* a small fraction of nodes. However, GES still achieves about 71.6 percent recall by probing *only* 30 percent nodes.
3. For the Reuters data, GES's performance is very close to that of SETS when probing less than 20 percent nodes. This is because the queries for the Reuters data are essentially *simple keyword match*.[7] GES can exactly identify a semantic group target node during search, thereby avoiding the overhead introduced by a "false" semantic group target node which unfortunately happened in the TREC data.
4. GES outperforms SETS when exploring more than 30 percent of the network for the TREC data and when exploring more than 20 percent of nodes for the Reuters data. We give the following explanations. First, the overhead of locating a right target node (and, thus, the semantic group) is amortized by probing more nodes. Second, the nature of GES's topology adaptation connects the *most* relevant nodes through *direct* semantic links, and it ensures a query probes the *most* relevant nodes first along

7. Search for documents containing all the keywords/terms in a query.

TABLE 2
The Distribution of Node Vector Sizes across 1,880 Nodes for TREC

Mean	*1st-percentile*	*99th-percentile*
1,776	88	8,099

semantic links. However, SETS does not distinguish the relevance between nodes within a topic segment and local links do not necessarily reflect that the *most* relevant nodes have *direct* connections. Therefore, some irrelevant nodes within topic segments are unavoidably visited when flooding the query within topic segments.

5. When exploring the whole network, the recall achieved by all three systems for the TREC data is 98.5 percent. This is because queries are short on average with only 3.5 terms in the TREC data. Some relevant documents could not be identified because their relevance scores computed using (1) are zero (note that simple keyword match does not need to compute relevance score and, thus, it will not miss identifying relevant documents for queries). During query evaluation, they are mistakenly deemed to be irrelevant due to such a low relevance score. In other words, with such short queries, the maximum recall achieved by a centralized IR system is 98.5 percent.

6.2 Effect of Node Vector Size

In the results presented for performance comparison, GES uses the *whole* node vector on each node to compute node relevance. To the best of our knowledge, SETS uses the whole node vector for node clustering. Since GES relies on node vector to compute node relevance, we believe *node vector size* could affect the quality of semantic groups and thus search performance. The node vector size, is defined as *the maximum number of terms considered in node relevance calculation per node vector*. Given a node vector $v = \{t_1, t_2, \ldots, t_{100}, \ldots, t_{500}\}$ with 500 terms where terms are in decreasing order of weight, the node vector size of 100 represents a node vector $v' = \{t_1, t_2, \ldots, t_{100}\}$, consisting of the 100 *top* terms which have the top-ranking (or heaviest) weight, while the node vector size of 800 will use the original full node vector v.

Due to the varying number of documents across nodes, the distribution of node vector sizes is skewed as shown in Table 2. In this section, we explore the effect of node vector size by conducting experiments to provide insight on the following questions: 1) What is a good node vector size in GES's performance, i.e., recall? 2) How could a substantial reduction in node vector size affect performance, i.e., recall?

Fig. 5 depicts the results for the TREC data. We observed similar characteristics on the results for the Reuters data. Fig. 5a plots the recall versus processing cost for various node vector sizes. Several observations can be drawn from this figure: 1) The node vector size of 1,000 performs the best, achieving 81 percent recall when probing only 30 percent of nodes. Table 3 summarizes the recall improvements made by the node vector size of 1,000 on SETS and GES (full), respectively. 2) The node vector size of 100 works very well, achieving about 68 percent recall when probing 30 percent of nodes. 3) The node vector sizes of 20 and 50, representing substantial reduction in node vector size, perform surprisingly well, achieving 63 percent and 67 percent recall respectively when probing 30 percent of nodes. Fig. 5b plots the cumulative distribution of recall for the queries with respect to the node vector size when probing 30 percent of nodes (we observed similar characteristics on other cases). Note that the node vector size of 1,000 outperforms other node vector sizes significantly.

From the above observations, a question is naturally raised: why does an appropriate node vector size (e.g., 1,000) perform the best while both a substantially larger (or even full) node vector size and a substantially smaller one degrade performance? To answer this question, we now look at Fig. 6.

For each document (vector), we sorted its terms in decreasing weight and calculated the relative weight of each term to the biggest term weight in the document. We then averaged this normalized term weight across all documents for each term rank, and plotted the mean for each term rank in Fig. 6a. The Y axis is in log scale. Note that the weight of the top 50 terms drops very fast. This confirms our intuition that a small number of terms (e.g.,

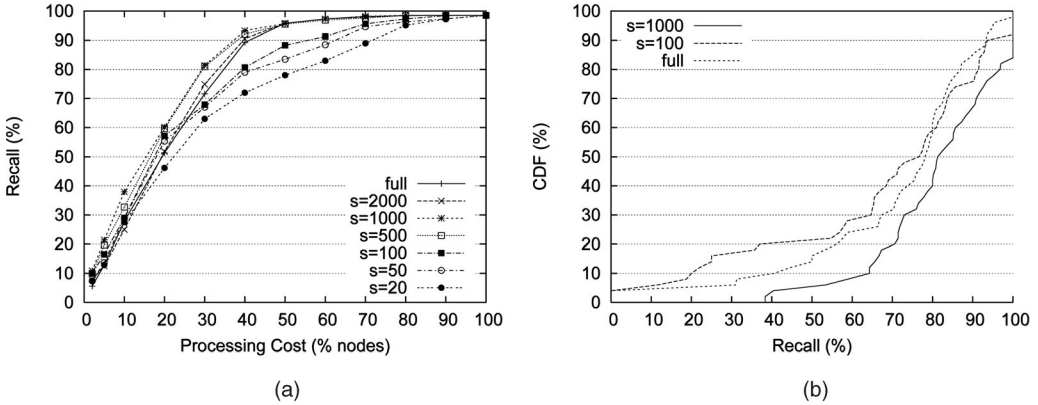


Fig. 5. (a) Recall versus processing cost for various node vector sizes. (b) Cumulative distribution of recall. s represents the node vector size and "full" means the whole node vector.

TABLE 3
The Recall Improvements with Respect to Query Processing Cost Made by GES (1,000) on SETS and GES (full), Respectively, for TREC

	processing cost (% nodes)						
	2%	*5%*	*10%*	*20%*	*30%*	*40%*	*≥50%*
GES(1000):SETS	-16.2%	-28.7%	-7.4%	-2.1%	8.5%	16.6%	≤6.8%
GES(1000):GES(full)	94.6%	67.2%	35.9%	17.1%	13.7%	4.5%	0%

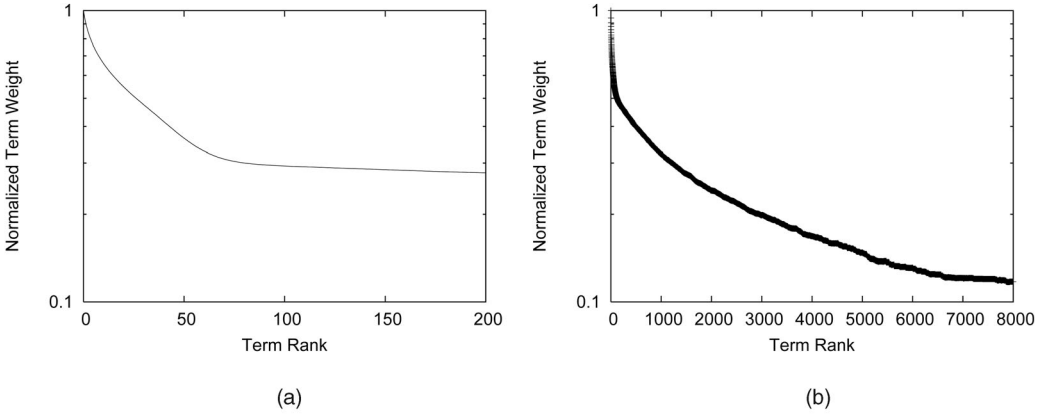


Fig. 6. (a) Ranked term weight for the TREC documents, normalized to the biggest term weight in each document. (b) Ranked term weight for the node vectors, normalized to the biggest term weight in each node vector.

50) are central to a document. They are capable of characterizing a document like the "Keyword" section of a technical paper.

Similarly, for each node vector, we sorted its terms in decreasing weight and calculated the relative weight of each term to the biggest term weight in the node vector. We then averaged this normalized term weight across all node vectors for each term rank, and plotted the mean for each term rank in Fig. 6b. The Y axis is in log scale. Note that the weight of the top 100 terms drops faster than Zipf distribution, and the weight of the top 1,000 terms also drops very fast. This shows that a relatively small number of terms are capable of characterizing a node's content because a node vector is derived from its documents, each of which is characterized by a small number of top terms (with heaviest weight).

As a result, a substantially small node vector size (e.g., 20 or 50) still performs very well since the node relevance score (according to (2)) is mostly determined by the top terms. However, a substantially small node vector size misses many important terms, and this degrades the performance of topology adaptation, which fails to identify some relevant nodes and put them within a semantic group, thereby impairing search performance. On the other hand, a substantially large node vector size (2,000, or even full size) contains too many (i.e., tens or even hundreds) unimportant terms which interfere with the calculation of the node relevance score—two nodes may be irrelevant even if their relevance score is high according to (2). We illustrate such interference through an example. Suppose two node vectors X and Y. They do not share top terms (say, 100 or 500), but they share many (say, tens or hundreds) unimportant terms. According to (2), their relevance score may be high enough to be mistakenly deemed to be relevant. This will negatively affect the quality of topology adaptation such that irrelevant nodes could be clustered into a semantic group. The node vector size of 1,000 strikes the balance. On the one hand, it catches most (or all) of the top terms, and on the other hand, it reduces the negative impact of tens or even hundreds of unimportant terms, thereby achieving the best performance.

An appropriate node vector size (e.g., 1,000) outperforms both a substantially larger (or even full) node vector size and a substantially smaller node vector size, because it more precisely characterizes node relevance. Compared to a larger node vector size, it also brings down the costs in *memory consumption (one-hop node vector replication), bandwidth usage (reduced message size during topology adaptation), and computation (the relevance score calculation consumes more time for larger node vector sizes)*. A significantly smaller node vector size (e.g., 20 or 50) can further bring down the aforementioned costs, while still showing pretty good performance. As a result, the node vector size exhibits a good design trade-off between search performance and resource consumption such as bandwidth cost. We leave the issue of how to dynamically determine an appropriate node vector size (e.g., through sampling) to our future work.

6.3 Flooding or Controlled Flooding?

The experiments so far assume that queries are flooded to *all* members within a semantic group. In this section, we investigate the effect of flooding radius on GES's performance. Recall that flooding originates from the target node of a semantic group, so the flooding radius is the distance (by hops) from the semantic group target node. With the flooding radius constraint, a query may visit a *fraction* of nodes within a semantic group (which we call *controlled flooding*). Fig. 7 plots the effect of flooding radius within semantic groups on GES's performance with the node

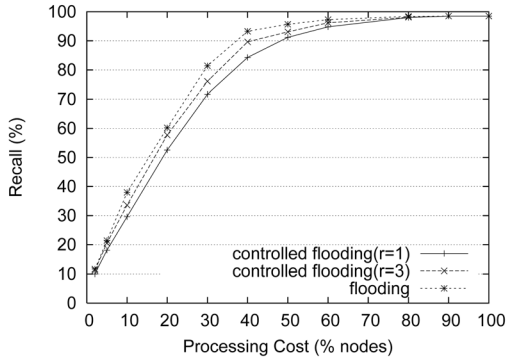


Fig. 7. Effect of flooding versus controlled flooding on recall for TREC. r represents the flooding radius within a semantic group for controlled flooding.

vector size of 1,000. Other node vector sizes also show similar characteristics. Due to space constraints, their results are omitted here.

Several observations can be drawn from this figure: 1) The controlled flooding with a radius of 1 performs very well. It achieves 71.7 percent recall when probing only 30 percent of nodes. This confirms that GES's topology adaptation connects the *most* relevant nodes with *direct* semantic links. Therefore, if a right target node is located, its most relevant neighbors will be visited to provide relevant documents for the queries. 2) The controlled flooding with a larger radius achieves better performance, and flooding the whole semantic group achieves the best performance. This confirms that the quality of semantic groups is good and nodes clustered within a semantic group tend to be relevant to the same queries. Thus, all the results presented in the rest of the paper are based on flooding rather than controlled flooding.

6.4 Topology Maintenance

GES relies on the periodically performed topology adaptation to maintain semantic groups and random links. On the one hand, each node every T seconds sends out two random walk queries to discover and update random and semantic neighbors; on the other hand, each node every T seconds checks with its current neighbors for their node vectors to remove disqualified neighbors. In this section, we examine the bandwidth cost associated with topology maintenance, by presenting both analytical and experimental results.

Suppose there are n nodes in the system. Assume an average of t terms per node vector, 4 bytes per term id, 4 bytes per term weight, and *no* compression of node vectors. We also assume an average of h hops per random walk query, an average of m neighbor candidates discovered for each random walk query, θ bytes for message overhead (e.g., TCP/IP header), and an average of d neighbors per node. On average, each random walk query consumes the overall bandwidth $b_q = (\theta + 8t) \times h$. The response messages corresponding to each random walk query consume bandwidth $b_r = (\theta + 8t) \times m$. Thus, every T seconds, each node consumes bandwidth $b_1 = 2(b_q + b_r)$ to discover new random and semantic neighbor candidates. To check with its current neighbors, each node sends a node vector request to a neighbor and receives a reply from the neighbor. So, every T seconds, each node consumes bandwidth $b_2 = (\theta + \theta + 8t) \times d$ to check with its d existing neighbors for fresh node vectors (including requests and responses). Moreover, each node every T seconds uses *keep-alive* messages to ensure the liveness of the neighbor candidates in its two neighbor caches. Assuming the cache size of s, the *keep-alive* messages consume bandwidth $b_3 = (\theta + \theta) \times 2s$. As a result, during T seconds, the overall bandwidth consumed by the topology adaptation across the system is $C = n \times (b_1 + b_2 + b_3)$. Then, the average bandwidth cost per node is $c = \frac{C}{n \times T}$ bytes/sec.

TABLE 4
The Average Bandwidth Cost per Node for Topology Maintenance during Four Hours of Simulation Times

Node vector size	500	1000	2000	full
Bandwidth cost (Kbps)	3.4	6.0	9.4	15.6

$T = 300$ *seconds*, $h = 10$, $m = 4$, $s = 100$, *and* $\theta = 20$ bytes. *No optimization is applied.*

Let us assume $\theta = 20$ bytes, $h = 10$, $m = 4$, $t = 1,000$ terms, $d = 8$, $s = 100$, and $T = 300$ seconds. We have the average bandwidth cost per node $c = 989.6$ bytes/sec. Put another way, we have $c = 7.9$ Kbps. The measurement study [34] has shown that about 65 percent of nodes in deployed P2P networks have low-speed access of 100 Kbps and 35 percent nodes have higher access speeds of 1.5 Mbps and 10 Mbps. Note that the topology adaptation consumes only 7.9 percent of its bandwidth for low-speed access nodes. Even for a node with dial-up access of 14.4 Kbps, the topology adaptation consumes only 55 percent of its bandwidth. Therefore, the bandwidth requirement for the topology adaptation is small. Note that the analytical number is in fact *pessimistic* since we do not apply any optimizations here. For example, during neighbor discovery, the response message corresponding to a random walk query can return the relevance score (i.e., 4 bytes) instead of the node vector (i.e., $8t$ bytes), thereby reducing the bandwidth cost b_r by several orders of magnitude. Only after the neighbor candidate is chosen as a neighbor will its node vector be transferred. When a node checks with its neighbors' node vectors, the node's neighbors can reply to the node with the *update delta* of their node vectors instead of the whole node vectors (e.g., most node vectors will not change in successive $T = 300$ seconds), thereby significantly reducing the cost b_2. Nodes within a semantic group can also exchange the node information in their semantic neighbor caches to reduce the neighbor discovery cost. Moreover, optimizations such as compression may further bring down the bandwidth cost. It is worth pointing out that a substantially small node vector size (e.g., 20 or 50) will greatly reduce the bandwidth cost while still showing pretty good performance (as discussed in Section 6.2).

Table 4 shows the experimental results for the average bandwidth cost per node with respect to the node vector size during four hours of simulation time. Note that bandwidth cost increases with the node vector size. The node vector size of 1,000 not only achieves the best search performance, but also incurs modest bandwidth overhead.

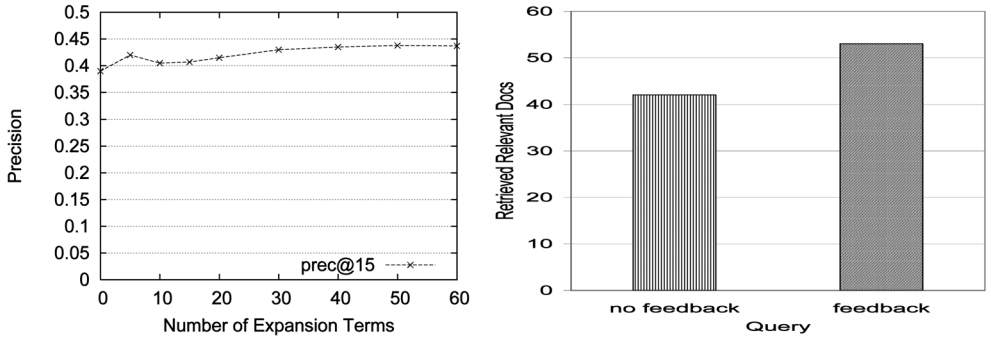


Fig. 8. (a) Precision@15 with respect to the number of added terms. The query processing cost is 30 percent nodes. (b) The average number of retrieved relevant documents for a query when retrieving 1,000 documents.

6.5 Automatic Query Expansion

Automatic query expansion [14] has been shown to be a very effective technique to improve precision and recall in centralized IR systems, especially for short queries. Assuming that the top few documents retrieved for the initial query are relevant, automatic query expansion adds relevant terms identified from the retrieved documents into the initial query to generate a new query. Then, the new query is fed into the system to retrieve the final set of results. Recall that in the TREC data, a query on average has 3.5 terms. Adopting automatic query expansion into GES may improve the quality of search results. In this section, we explore the effect of automatic query expansion on precison@15 and recall with a set of 38 queries (the other 12 queries were excluded from this experiment because each has less than 15 relevant documents). It is worth pointing out that the queries in the Reuters data are simple keyword match and, thus, do not need to do automatic query expansion.

Given a query, GES first retrieved a small number k of most relevant documents. It used these documents as *feedback documents*. GES computed the average weight of each term appearing in the feedback documents and chose n top terms which have the heaviest weight. These n terms were assumed to be relevant to the query and were added to expand the query (the weight of each of these terms were divided by a constant c). The new query was then used to retrieve the final set of relevant documents. Our experimental results showed $k = 10$ and $c = 16$ worked very well. Fig. 8a shows automatic query expansion improves precision, but not significantly. For example, when the number of added terms is 30, the improvement on precision@15 is about 10 percent. Fig. 8b shows query expansion improves recall by about 26 percent (the number of added terms is 30).

6.6 Local Data Clustering (LDC)

As shown in Section 5.2, the documents on a node are diverse. A question is naturally raised: can we improve the quality of semantic groups and, thus, search performance by distinguishing diverse topics in a node's documents? To answer this question, we introduce a notion of *virtual node* [1]. A node with diverse-topic documents locally clusters its documents using data clustering techniques and each cluster corresponds to a virtual node. A node with diverse-topic documents thus may host multiple virtual nodes, each of which independently participates in GES's topology adaptation and search protocol.

Clustering documents on a node can use many data clustering techniques (e.g., hierarchical or k-means), each of which may give a different group of a data set. We leave the issue of exploring impact of different clustering techniques to our future work. For simplicity, we assume only those nodes with a certain number of documents (e.g., ≥ 20) needs to do LDC. This results in a total of about 38 percent nodes (out of 1,880 nodes) which need to do LDC. The number of clusters on a node is predefined by the number of queries for which the node provides relevant documents. This is based on the observation that the queries are diverse in topics. Each cluster corresponds to a virtual node and its *center* represents the node vector of the virtual node.

To avoid unnecessarily biasing against the design without LDC, we keep unchanged the maximum number of random links per *physical* node in the presence of virtual nodes. Otherwise, a node may have more random links by hosting virtual nodes, thereby improving the performance of biased walks and, thus, overall search performance. When forming random links, the node vector of a *physical* node is involved; when forming semantic links, however, the node vector of a *virtual* node is involved. As a result, virtual nodes here are just used to improve quality of semantic groups.

Fig. 9 depicts the cumulative distribution of recall without LDC for queries when probing 30 percent of nodes. Two main observations can be drawn from this figure: 1) "full+ldc" performs better than "full." This shows LDC improves the quality of semantic groups, thereby achieving better performance. 2) "1,000+ldc" shows not much improvement over "1,000." We give the following explanations. First, the node vector size of 1,000 is already capable of precisely characterizing the relevance between nodes, thereby producing high quality of semantic groups. Second, the number of documents per node is not very large. Recall that the average number of documents per node is 42.5 and the 99th percentile is 417. We expect that LDC would perform much better in the scenarios where the number of documents per node is much larger and the topics are more diverse.

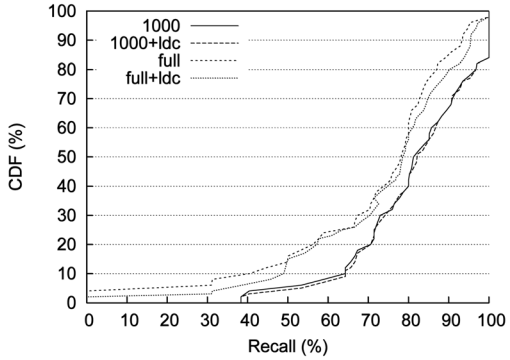


Fig. 9. Cumulative distribution of recall for TREC. "full+ldc" represents using full node vector size and local data clustering. "1,000+ldc" represents using the node vector size of 1,000 and local data clustering.

6.7 Impact of Heterogeneity

So far, all our experiments have been conducted by assuming uniform node capacity. In this section, we explore the performance gain achieved by GES's capacity-aware mechanism, which exploits node heterogeneity. Based on a Gnutella-like profile [34], we assigned capacity of $1\times$, $10\times$, $10^2\times$, $10^3\times$, and $10^4\times$ to nodes with probability of 20 percent, 45 percent, 30 percent, 4.9 percent, and 0.1 percent, respectively. Nodes with capacities $1,000\times$ and $10,000\times$ were assumed to be supernodes.

Fig. 10 shows the performance of GES with heterogeneous and uniform capacity. "Heter" represents the scenario where node capacity is heterogeneous while "Uniform" represents the scenario where node capacity is uniform. The number in parenthesis is the node vector size and "full" represents the full node vector size. Note that exploiting node capacity heterogeneity can significantly improve performance (e.g., achieving 73 percent recall by probing only 20 percent of nodes). Table 5 summarizes the recall improvements made by GES (1,000) on SETS, which does not have a capacity-aware mechanism.

7 Conclusions and Future Work

GES is our first attempt at enhancing search performance in terms of efficiency and quality of search results on Gnutella by exploiting IR algorithms. Our main contributions are:

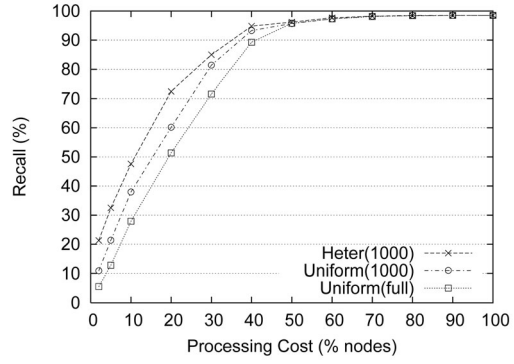


Fig. 10. Effect of node capacity heterogeneity on recall for TREC.

TABLE 5
The Recall Improvements with Respect to Query Processing Cost Made by GES (1,000) on SETS in the Scenario Where Node Capacity is Heterogeneous

	processing cost (% nodes)						
	2%	*5%*	*10%*	*20%*	*30%*	*40%*	*≥50%*
GES(1000):SETS	63.8%	8.3%	16.1%	17.9%	13.3%	18.5%	≤7.4%
GES(1000):SETS	64.7%	9.6%	18.3%	18.6%	13.7%	18.1%	≤7.3%

GES (1,000) represents the case where GES uses the node vector size of 1,000. The first row represents improvements for TREC and the second row represents improvements for Reuters.

1. We use a fully distributed topology adaptation algorithm to restructure overlay topology for search performance enhancement while retaining the simple, robust, and fully decentralized nature of Gnutella. Both analytical and experimental results show that the bandwidth cost of topology adaptation is modest.
2. GES is the first to show that the node vector size plays an important role in search performance. We found that the node vector size exhibits a good design trade-off between search performance and bandwidth cost. For example, to reduce bandwidth cost, GES can use a substantially small node vector size (i.e., 20) while still showing pretty good performance (i.e., achieving 63 percent recall when probing only 30 percent nodes).
3. GES employs IR algorithms such as automatic query expansion and local data clustering to improve search performance.
4. GES's capacity-aware mechanism can exploit node heterogeneity to enhance performance. Via simulations, we show that GES outperforms the centralized node clustering system SETS. For example, in the system where node capacity is heterogeneous, GES can achieve 73 percent recall when probing only 20 percent nodes, outperforming SETS by about 18 percent.

Several issues need to be addressed in our future work. First, we need to examine the impact of data replication on search performance because the data sets currently used in our experiments do not have the characteristic of data replication. Second, we will explore other IR algorithms such as LSI in GES design. Finally, we plan to investigate the impact of various data clustering techniques on search performance.

Acknowledgments

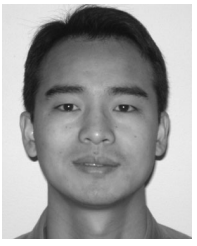
The authors thank the anonymous reviewers for their helpful comments and suggestions, which greatly improved the final version of this paper. They also thank Mayank Bawa for the discussions on SETS.

References

[1] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGCOMM*, pp. 149-160, Aug. 2001.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Proc. 18th IFIP/ACM Int'l Conf. Distributed System Platforms (Middleware)*, pp. 329-350, Nov. 2001.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM SIGCOMM,* pp. 161-172, Aug. 2001.
[4] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerance Wide-Area Location and Routing," Technical Report UCB/CSD-01-1141, Computer Science Division, Univ. of California, Berkeley, Apr. 2001.
[5] J. Li, B.T. Loo, J. Hellerstein, F. Kaashoek, D.R. Karger, and R. Morris, "On the Feasibility of Peer-to-Peer Web Indexing and Search," *Proc. Second Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 207-215, Feb. 2003.
[6] P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *Proc. ACM/IFIP/USENIX Int'l Middleware Conf. (Middleware),* pp. 21-40, June 2003.
[7] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks," *Proc. ACM SIGCOMM,* pp. 175-186, Aug. 2003.
[8] Y. Zhu, H. Wang, and Y. Hu, "Integrating Semantics-Based Access Mechanisms with P2P File Systems," *Proc. Third Int'l Conf. Peer-to-Peer Computing,* pp. 118-125, Sept. 2003.
[9] Q. Lv, P. Cao, and E. Cohen, "Search and Replication in Unstructured Peer-to-Peer Networks," *Proc. 16th ACM Ann. Int'l Conf. Supercomputing (ICS),* pp. 84-95, June 2002.
[10] E. Cohen, H. Kaplan, and A. Fiat, "Associative Search in Peer to Peer Networks: Harnessing Latent Semantics," *Proc. IEEE INFOCOM,* vol. 2, pp. 1261-1271, Apr. 2003.
[11] A. Crespo and H. Garcia-Molina, "Routing Indices for Peer-to-Peer Systems," *Proc. 22nd IEEE Int'l Conf. Distributed Computing Systems (ICDCS),* pp. 23-32, July 2002.
[12] K. Spripanidkulchai, B. Maggs, H. Zhang, "Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems," *Proc. IEEE INFOCOM,* vol. 3, pp. 2166-2176, Mar. 2003.
[13] M. Bawa, G. Manku, P. Raghavan, "SETS: Search Enhanced by Topic Segmentation," *Proc. 26th Ann. Int'l ACM SIGIR Conf.,* pp. 306-313, July 2003.
[14] M. Mitra, A. Singhal, C. Buckley, "Improving Automatic Query Expansion," *Proc. ACM SIGIR,* pp. 206-214, 1998.
[15] C. Tang, S. Dwarkadas, "Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval," *Proc. USENIX/ACM Symp. Networked Systems Design and Implementation (NSDI),* Mar. 2004.
[16] C. Tang, S. Dwarkadas, Z. Xu, "On Scaling Latent Semantic Indexing for Large Peer-to-Peer Systems," *Proc. 27th Ann. Int'l ACM SIGSIR Conf.,* July 2004.
[17] N. Ntarmos, P. Triantafillou, "AESOP: Altruism-Endowed Self-Organizing Peers," *Proc. Second Int'l Workshop Databases, Information Systems, and Peer-to-Peer Computing,* pp. 151-165, Aug. 2004.
[18] A. Gupta, B. Liskov, R. Rodrigues, "Efficient Routing for Peer-to-Peer Overlays," *Proc. First Symp. Networked Systems Design and Implementation (NSDI),* Mar. 2004.
[19] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, "Handling Churn in a DHT," *Proc. 2004 USENIX Technical Conf.,* June 2004.
[20] C.H. Ng and K.C. Sia, "Peer Clustering and Firework Query Model," *Proceedings World Wide Web Conf. (WWW),* May 2002.
[21] P. Triantafillou, C. Xiruhaki, M. Koubarakis, N. Ntarmos, "Toward High Performance Peer-to-Peer Content and Resource Sharing Systems," *Proc. CIDR,* Jan. 2003.
[22] F.M. Cuenca-Acuna, C. Peery, R.P. Martin, T.D. Nguyen, "PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," *Proc. 12th IEEE Int'l Symp. High Performance Distributed Computing (HPDC),* June 2003.
[23] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, "Oceanstore: An Architecture for Global-Scale Persistent Storage," *Proc. ACM ASPLOS,* Nov. 2000.
[24] B.T. Loo, R. Huebsch, I. Stoica, J.M. Hellerstein, "The Case for a Hybrid P2P Search Infrastructure," *Proc. Third Int'l Workshop Peer-to-Peer Systems (IPTPS),* Feb. 2004.
[25] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker, "Making Gnutella-Like P2P Systems Scalable," *Proc. ACM SIGCOMM,* pp. 407-418, Aug. 2003.
[26] T. Hernandez, S. Kambhampati, "Improving Text Collection Selection with Coverage and Overlap Statistics," *Proc. 14th Int'l World Wide Web Conf. (WWW),* May 2005.
[27] M. Bender, S. Michel, P. Triantafillou, G. Weikum, C. Zimmer, "Improving Collection Selection with Overlap Awareness in P2P Search Engines," *Proc. 28th Int'l ACM SIGIR Conf.,* Aug. 2005.
[28] M.W. Berry, Z. Drmac, E.R. Jessup, "Matrices, Vector Spaces, and Information Retrieval," *SIAM Rev.,* vol. 41, no. 2, pp. 335-362, 1999.
[29] H. Schutze, C. Silverstein, "A Comparison of Projections for Efficient Document Clustering," *Proc. ACM SIGIR,* pp. 74-81, July 1997.
[30] R. Lempel, S. Moran, "Optimizing Result Prefetching in Web Search Engines with Segmented Indices," *Proc. Conf. Very Large Data Bases (VLDB),* 2001.
[31] *Proc. Text Retrieval Conf. (TREC),* http://trec.nist.gov, July 2005.
[32] Reuters Corpus, Nov. 2000, http://www.reuters.com/research andstandards/corpus.
[33] C. Buckley, "Implementation of the Smart Information Retrieval System," Technical Report TR85-686, Dept. of Computer Science, Cornell Univ., May 1985.
[34] S. Saroiu, K.P. Gummadi, and S.D. Gribble, "Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts," *Multimedia Systems J.,* vol. 9, pp. 170-184, Aug. 2003.



Yingwu Zhu received the PhD degree in computer science and engineering from the University of Cincinnati in 2005. He received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, China, in 1994 and 1997, respectively. He is an assistant professor of computer science and software engineering at the Seattle University. His research interests include operating systems, file and storage systems, peer-to-peer systems, distributed systems, and sensor networks.

Yiming Hu received the PhD degree in electrical engineering from the University of Rhode Island in 1998. He received the BE degree in computer engineering from the Huazhong University of Science and Technology, China. He is an associate professor of computer science and engineering at the University of Cincinnati. His research interests include computer architecture, storage systems, peer-to-peer systems, operating systems, and performance evaluation. He is a recipient of a US National Science Foundation CAREER Award. He is a senior member of the IEEE.