

Adaptive Probabilistic Search for Peer-to-Peer Networks

Dimitrios Tsoumakos
Department of Computer Science
University of Maryland
dtsouma@cs.umd.edu

Nick Roussopoulos
Department of Computer Science
University of Maryland
nick@cs.umd.edu

Abstract

Peer-to-Peer networks are gaining increasing attention from both the scientific and the large Internet user community. Popular applications utilizing this new technology offer many attractive features to a growing number of users. At the heart of such networks lies the search algorithm. Proposed methods either depend on the network-disastrous flooding and its variations or utilize various indices too expensive to maintain. In this paper, we describe an adaptive, bandwidth-efficient algorithm for search in unstructured Peer-to-Peer networks, the Adaptive Probabilistic Search method (APS). Our scheme utilizes feedback from previous searches to probabilistically guide future ones. It performs efficient object discovery while inducing zero overhead over dynamic network operations. Extensive simulation results show that APS achieves high success rates, increased number of discovered objects, very low bandwidth consumption and adaptation to changing topologies.

1. Introduction

Peer-to-Peer (hence P2P) networking has been growing rapidly in the last few years. It represents the notion of sharing resources available at the edges of the Internet. Its success was originally boosted by some very popular file-sharing applications (e.g., [11]). Numerous systems that utilize or support P2P technology have emerged since (e.g., [12, 16]). The P2P paradigm dictates a fully-distributed network design which exhibits robustness in failures, extensive resource-sharing, self-organization, load balancing, data persistence, anonymity, etc.

We can roughly classify P2P architectures into two categories: *Centralized* approaches utilize a central directory for object location, ID assignment, etc. *Decentralized* approaches can either follow the *pure* model, with all peers (or nodes) equally making, routing and answering requests, or a *hybrid* one, where peers are divided into *leaf-nodes* and *super-peers*. The latter serve requests for their neighboring leaf-nodes and shield them from the rest of the

network. Another taxonomy classifies P2P networks into *structured* and *unstructured*. While *structured* networks provide strict rules for file placement and object discovery, the most popular P2P applications operate on *unstructured* networks. These approaches offer arbitrary network topology, file placement and search. Bandwidth consumption attributed to P2P file-sharing systems amounts to a considerable fraction (up to 60%) of the total Internet traffic [15]. Moreover, popular P2P networks display a highly dynamic behavior, with most users connecting for small periods of time and then leaving the system [2]. Therefore, it is vital that search schemes be both bandwidth-efficient and *robust* (showing fault-tolerance and adaptation in dynamic environments).

A search for an object in a P2P network is *successful* if it discovers at least one replica of the object. The ratio of successful to total searches made is the *success rate* (or *accuracy*) of the algorithm. A search can result to multiple discoveries (or *hits*), which are copies of the same object stored at *distinct* nodes. *Duplicate* messages are copies of the same query sent to a node that has already processed it. The *performance* of an algorithm is associated with its success rate and number of hits, while its *cost* relates to the number of messages it produces.

Search methods can be categorized as either *blind* or *informed*. In a *blind* search, nodes do not store any information regarding file locations. In *informed* approaches, nodes locally store metadata that assist in the search for the queried objects. Current blind methods waste a lot of bandwidth to achieve high performance. Every search requires contacting many nodes within some distance called *time-to-live (TTL)*, generating huge overhead to all nodes involved. Informed methods use their indices in order to achieve similar quality results (by choosing “good” neighbors to forward the query to) and to reduce overhead. The shortcoming of most informed methods is the maintenance cost of the indices after peers join/leave the network or update their collections. In most cases, these events trigger *floods* of update messages, inflating network traffic.

Many search protocols for *unstructured* networks have

been proposed with an intention to reduce the overhead of the original Gnutella *flooding* scheme [6] (breadth-first traversal of the underlying graph). In the *Random Walks* method [9], the requesting node sends out k query messages to an equal number of randomly chosen neighbors. Each of these queries follows its own path, having intermediate nodes forward it to a randomly chosen neighbor at each step. These queries are known as *walkers*. While this approach manages to reduce messages by more than an order of magnitude, it exhibits low performance due to its random nature and inability to adapt to different query loads.

In this paper, we propose a new search algorithm that achieves high performance at low cost, the *Adaptive Probabilistic Search* method (*APS*). In *APS*, a node deploys k walkers for object discovery, but the forwarding process is probabilistic instead of random. Peers effectively direct walkers using feedback from previous searches, while keeping information only about their neighbors. As we show in this work, *APS* exhibits many plausible characteristics, namely high accuracy, low bandwidth consumption, large number of discovered objects and robust behavior in fast-changing environments. These features come as a result of our algorithm's *learning* character, which enables peers to share, refine and adjust their search knowledge with time. Furthermore, *APS* induces zero overhead over the network at join/leave/update operations.

This paper makes the following contributions:

1. We define the *APS* algorithm for search in unstructured P2P networks. We describe the main idea, the indexing scheme, the search and update procedures and analyze its performance.
2. We present two improved versions of the algorithm which exhibit significant gains in message reduction and the number of objects discovered near the requesters.
3. We perform extensive simulations and compare *APS* with the methods in [4, 9] over different environments. Our algorithm achieves great results in the success rate, message production, number of hits and adaptation to changing topologies.

2. Related Work

Structured P2P systems base all operations on an “overlay” network, which handles file and replica placement and guarantees bounded number of steps and reliable storage. Examples of such systems include [13, 14, 17]. They perform very efficient searches but incur big overheads during peer join/leave operations.

Many search algorithms for unstructured P2P networks have also been proposed in the last few years. Ref. [5] presents a thorough categorization and description of many approaches. Ref. [8] proposes a variation of the flooding

scheme with peers randomly choosing only a ratio of their neighbors to forward the query to. The same procedure is followed in [10] by nodes with no information about the location of a file. If an object is found, the query takes the reverse path to the requester, storing the document location at those peers. Nodes with location information contact the specific node directly.

Two new protocols for Gnutella-type networks operate on hybrid overlays. In *GUESS* [4], searches are conducted by iteratively contacting different super-peers and having them ask all their leaf-nodes, until some predefined condition is met. In Gnutella2 [18], a super-peer that receives a query from a leaf forwards it to its relevant leaves and to its neighboring super-peers. The latter forward it to their relevant leaves. No other nodes are visited with this algorithm.

In [1], the degrees of the nodes are used to guide walkers in a power-law graph. In contrast, our algorithm uses hints that relate to search results, aiming for both performance enhancement and knowledge build-up.

Several informed methods have also been proposed. In [8], each node forwards keyword requests to a set number of neighbors that have answered the most requests “similar” to the current one, according to a query similarity metric. Nodes store information about recently answered queries in order to rank their neighbors. Our approach utilizes both positive and negative feedback from the walkers so that efficient unlearning is performed, while neighbors are probabilistically chosen, not ranked.

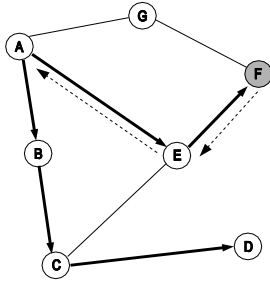
In [19], a node holds information about all files stored at nodes within a certain radius and can answer queries on behalf of all of them. In [3], nodes store file content metadata for each of their outgoing links, enabling them to forward a query to the neighbor with the highest value of a defined metric. In our approach, nodes keep indices regarding only their neighbors, avoiding the cost of updates at every change. Furthermore, our index semantics relate to previous search results, not file locations.

3. The *APS* Algorithm

3.1. Our Search Model

The following assumptions are made throughout the discussion that follows:

Peers initiate searches for various objects. These objects are distributed across the network according to a *replication distribution*, which dictates what objects are stored at each node. The *query distribution* dictates how many requests are made for each object (e.g., popular objects get many more requests than unpopular ones). The search algorithms cannot in any way dictate object placement and replication in the system. They are also not allowed to alter the topology of the P2P overlay. A node is directly connected to its *neighbors*, and these are the only peers whose addresses the



Indices	Initially	After walkers finish	After the updates
A→B	30	20	20
B→C	30	20	20
C→D	30	20	20
A→E	30	20	40
E→F	30	20	40
A→G	30	30	30

Figure 1. Search for an object stored at node F using the pessimistic approach of APS with two walkers. The table shows how various index values change, where X→Y denotes the index value stored at node X for neighbor Y relative to the requested object.

node is always aware of.

Nodes can keep some *soft state* (i.e., information that is erased after a short amount of time) for each query they process. Each search is assigned an identifier, which, together with the soft state, enables peers to make the distinction between new and duplicate messages. Identifiers are also assigned to objects and nodes from a flat, non-hierarchical space. The *TTL* parameter represents the maximum hop-distance a query can reach before it gets discarded, while *k* denotes the number of walkers deployed for search from a requester node.

Finally, the main metrics we use to evaluate the performance of a search algorithm are the success rate, the number of discovered objects and the number of messages produced. For simplicity reasons, we ignore network and processing delays. While such delays affect response time, they cannot impact our metrics.

3.2. Algorithm Description

In *APS*, each node keeps a local index consisting of one entry for each object it has requested, or forwarded a request for, per neighbor. The value of each entry reflects the relative probability of this node's neighbor to be chosen as the next hop in a future request for the specific object.

Searching is based on the simultaneous deployment of *k* walkers and probabilistic forwarding: The requester chooses *k* out of its *N* neighbors (if $k \geq N$, the query is sent to all neighbors) to forward the request to. Each of these nodes evaluates the query against its local repository and if a hit occurs, the walker terminates successfully. On a miss, the query is forwarded to one of the node's neighbors. This procedure continues until all *k* walkers have terminated, either with a success or a failure. So, while the requesting node forwards the query to *k* neighbors, the rest of the nodes forward it to only one. In the forwarding process, a node chooses its next-hop neighbor(s) not randomly, but using the probabilities given by its index values. At each forwarding step, nodes append their identifiers in the

search message and keep a soft state about the search they have processed. If two walkers from the same request cross paths (i.e., a node receives a duplicate message due to a cycle), the second walker is assumed to have terminated with a failure and the duplicate message is discarded.

Index values stored at peers are updated in the following manner: When a node forwards the request to one or *k* of its neighbors, it pro-actively either increases the relative probability of the peer(s) it picked, assuming the walker(s) will be successful (*optimistic* approach), or it decreases the relative probability of the chosen peer(s), assuming the walker(s) will fail (*pessimistic* approach).

Upon walker termination, if the walker is successful, there is *nothing* to be done in the *optimistic* approach. If the walker fails, index values relative to the requested object along the walker's path must be corrected. Using information available inside the search message, the last node in the path sends an "*update*" message to the preceding node. This node, after receiving the update message, *decreases* its index value for the last node to reflect the failure. The update procedure continues along the reverse path towards the requester, with intermediate nodes decreasing their local index values relative to the next hops for that walker. Finally, the requester decreases its index value that relates to its neighbor for that walker. If we employ the *pessimistic* approach, this update procedure takes place after a walker succeeds, having nodes increase the index values along the walker's path. There is nothing to be done when a walker fails.

Figure 1 shows an example of how the search process works. Node A initiates a request for an object owned by node F using two walkers. Assume that all index values relative to this object are initially equal to 30 and the *pessimistic* approach is used. The paths of the two walkers are shown with thicker arrows. During the search, the index value for a chosen neighbor is reduced by 10. One walker with path (A,B,C,D) fails, while the second with path (A,E,F) finds the object. The update process is initi-

ated for the successful walker on the reverse path (along the dotted arrows). First node E, then node A increase the value of their indices for their next hops (nodes F, E respectively) by 20 to indicate object discovery through that path. In a subsequent search for the same object, peer A will choose peer B with probability $2/9$ ($=\frac{20}{20+40+30}$), peer E with probability $4/9$ and peer G with probability $3/9$.

Our method utilizes “probabilistic” walkers with a *learning* feature that incorporates knowledge from past and present searches to enhance future performance. The learning process adaptively directs the walkers to promising parts of the network, while keeping bandwidth consumption low.

APS requires no message exchange on any dynamic operation such as node arrivals or departures and object insertions or deletions. The nature of the indices makes the handling of these operations simple: If a node detects the arrival of a new neighbor, it will associate some initial index value with that neighbor when a search will take place. If a neighbor disconnects from the network, the node removes the relative entries and stops considering it in future queries. No action is required after object updates, since indices are not related to file content. So, although our algorithm actively uses information, its maintenance cost on any of these events is zero, a major advantage over most current approaches.

3.3. Discussion

Each node stores a relative probability (e.g., an unsigned integer value) for each of its neighbors for each (directly or indirectly) requested object. So, for \mathcal{R} such objects and N neighbors, $O(\mathcal{R} \times N)$ space is needed. For a typical network node, this amount of space is not a burden. On nodes with limited storage capacities, index values for objects not requested for some time can be erased. This can be achieved by assigning a time-to-expire value on each newly-created or updated index, or by expunging the least recently (or frequently) used indices. Each search or update message carries path information, storing a maximum of *TTL* peer addresses. Alternatively, each node can associate the search and requester node IDs with the preceding peer in the path of the walker. Updates then follow the reverse path back to the requester. This information expires after a certain amount of time. A selection from the above techniques depends on the application, query workload and node capabilities.

Let us calculate how many messages it will take for the *APS* method to terminate. In the worst case — all walkers travel *TTL* hops and then invoke the update procedure — the number of messages exchanged will be $2 \times k \times TTL$, so the method has the same complexity with its random counterpart. The only extra messages that occur in *APS* are the update messages along the reverse path. This is where our two index update policies are used: If we expect or expe-

rience after a while that for a specific number of walkers k , only few of them terminate successfully, then the *pessimistic* mode should be employed. Conversely, if many of our walkers hit their targets on average, the *optimistic* approach should be considered. Naturally, the two approaches have the same performance in all other metrics.

Along the paths of all k walkers, indices are updated so that better next hop choices are made with bigger probability. Our learning feature includes both positive and negative feedback from the walkers in both update approaches. In the *pessimistic* approach, each node on the walker’s path decreases the relative probability of its next hop for the requested object concurrently with the search. If the walker succeeds, the update procedure increases those index values by more than the subtracted amount (positive feedback). So, if the initial probability of a node for a certain object was P , it becomes bigger than P if the object was discovered through (or at) that node and smaller than P if the walker failed. This is the only invariant we require from our update process. In the next section, we compare several functions with this characteristic. The learning process in the *optimistic* approach operates in an opposite fashion, with negative feedback taking place after a walker fails. Our algorithm exhibits both *learning* and *unlearning* characteristics: *Learning* is important to achieve both high performance and discovery of newly inserted objects. *Unlearning* helps our search process adjust to object deletions and node departures, redirecting the walkers elsewhere.

Another characteristic of the algorithm is its ability to obtain more knowledge with more questions. The more feedback from the walkers, the more precise the indices become. That particularly suits the discovery of popular objects in the P2P network, which, according to studies [2], constitute over 60% of all searches. Another similar observation is that all nodes participating in a search will benefit from the process. This is a distinctive feature of our method, with indices being constantly updated using search results and not after object updates. In our case, *both* requesters and peers on the paths of all walkers actively adjust their knowledge about the specific object. A node that has never before requested an object but is “near” peers that have done so, inherits this knowledge by proximity. Besides standard resource-sharing in P2P systems, our algorithm achieves the distribution of *search knowledge* over a large number of peers.

3.4. Algorithm Improvements

APS produces update messages to adjust index values along the paths of some walkers. Our goal is to minimize these messages in order to further reduce the level of bandwidth consumption. Obviously, if fewer than $k/2$ walkers are successful, then the *pessimistic* approach should be employed instead of the *optimistic* and vice versa. Choosing

one strategy over the other for queries over all objects is not optimal, as many unnecessary update messages would be produced for both popular and unpopular object requests. In *swapping-APS* (*s-APS*), the algorithm constantly monitors the ratio of successful walkers for each request and accordingly switches to the update policy that produces fewer messages. This makes our *s-APS* improvement more bandwidth efficient, sometimes producing a lower total number of messages *even* from *Random Walks*, which uses no indices. The number of requests for which nodes monitor the successful walker ratio depends on available node storage, although the overhead will be very small in most cases.

Another improvement relates to the index update procedure. The idea is to give preference to the location of objects located near the requester nodes. Our *weighted* approach (*w-APS*) incorporates a distance-based function for modifying the relative probabilities stored at each node. Index values for peers closer to the discovered object are increased more than those for distant nodes. Distance information is directly accessible from the stored path inside the search messages. With this method, peers are biased to direct walkers to objects that are near. Our results show a significant increase in the number of discovered objects located near the requesters.

Both improved versions impose no extra burden to the search process, while they aim at reducing its average response time (either with smaller message production or location of nearby objects).

4. Simulation Results

4.1. Simulation Methodology

To simulate the P2P overlay, we mainly used the *random* graph topology with the *pure* P2P model. We also experimented on the popular *power-law* topology as well as the *hybrid* model for a comparison with *GUESS*. GT-ITM [20] was utilized for the pure and hybrid random graph models and Inet-3.0 [7] for the power-law graph model.

For the object replication and query strategies, we choose from three different distributions, namely uniform, zipf and 80/20. Requesters are randomly chosen and always represent a noticeable fraction (around 10%) of the size of the graph. The default graph has 10,000 nodes with an average out-degree $d = 10$. The default values for k and TLL are 12 and 6 respectively.

To simulate a dynamic network behavior, we insert “on-line” nodes and remove active ones with varying frequencies. In the first setting (static), there are no dynamic operations. In the less dynamic setting, the topology changes more than 300 times during each run, while in the more dynamic one it changes 10 times more frequently. We always keep approximately 80% of the network nodes active.

Departing nodes clear their local cache from all built knowledge.

We used 100 objects in most simulations for simplicity and speed. Objects are of varying popularity, which affects the respective number of replicas and received requests. An increase in the number of objects did not affect the quality of the results. We modeled the query and replication strategies using a zipfian distribution to achieve results similar to the observations in [2]. The highest-ranked 10% of objects amount to about 50% of the total number of stored objects and receive about half of the requests. Results for other skewed distributions (e.g., 80/20) are qualitatively similar. With our default parameters, the most popular object is stored in more than 10% of the peers, while the least popular only in 0.25% of them.

In the figures that follow, the label “*APS*” is used when all variations of our method have very similar performance in a particular metric. If the results were taken under any of the two dynamic settings, this will be shown in parenthesis.

4.2. Choosing the index update functions

In our first set of simulations, we try to evaluate the performance of several index update functions towards achieving fast unlearning and discovering the nearest objects (used with *w-APS*).

On a random graph, we made queries with $k = 1$ from a certain node. Only one of its neighbors initially obtained a replica of the object. After only 10 queries, we removed this replica and inserted another one into a random node 3 hops away. Note that this is more challenging than simply removing the node with the initial replica. Keeping the node active forces *APS* to consider it for future requests with the already accumulated index knowledge. We monitored the accuracy achieved by several functions after the deletion and present the results for four of them in Figure 3. We evaluated a flat update function (change indices by a set value each time), a step function (amount of change depends on the range of the index value), a linear function (amount of change is a linear function of the current index value) and a hop-count related function (as described for the *w-APS* method).

The dotted line represents the accuracy of *APS* before the replica relocation (10 queries). We notice that, while all functions “learn” with more queries, the linear function increases its accuracy faster to match the initial success rate. Both the linear and the step function that approximates it show that unlearning is more effective if the amount of index *decrease* is proportional to its value. Obviously, the rate at which nodes (and therefore paths to objects) depart affects the efficiency of the unlearning process.

In a similar experiment for the *w-APS* method, we monitor the percentage of hits for each replica. The flat/linear/step functions found the nearest object about 60% of the time. A function with the amount of increase being

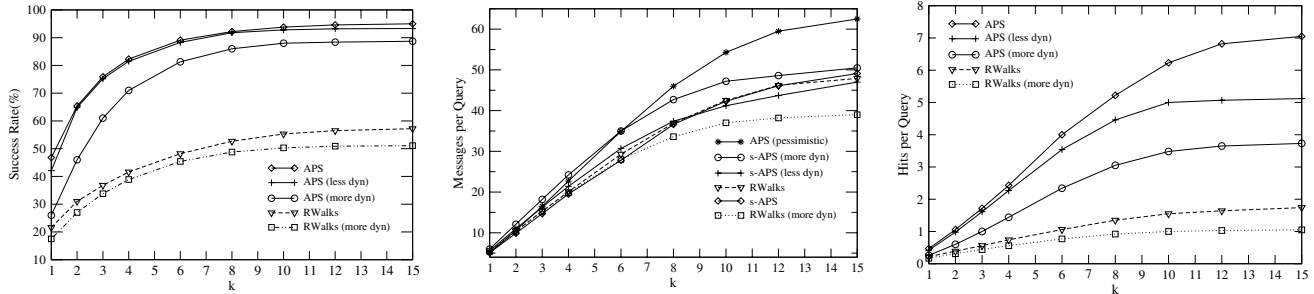


Figure 2. Success rate, message production and number of hits of the two methods vs. number of deployed walkers in the three different settings

inversely proportional to the distance raised to the power of 2 discovered the nearest replica over 90% of the time. This function will be used to evaluate the *w-APS* version.

4.3. APS performance and comparison with Random Walks

In the next set of simulations, we try to validate the analysis of Section 3. We vary the number of walkers deployed (k) from 1 to 15 for the default parameters and test the two algorithms on all three settings. Figure 2 presents the detailed comparison on the three important metrics (accuracy, hits and messages per query).

Random Walks exhibits low performance as a result of its nature. Its accuracy is below 50% for most simulations, while it barely averages one hit per query in the dynamic settings, even with many walkers. Its message production is further reduced during the dynamic runs, since long paths inside the network (taken by the method's unsuccessful walkers) frequently disappear. The performance decrease during the dynamic runs is relatively small, as walkers are not directed according to object locations, but randomly across the network. For that reason, we omit its results relative to the less dynamic case.

On the other hand, *APS* achieves high quality results in all these metrics. The trade-off for using its adaptive scheme is the performance decrease in the dynamic settings. *APS* manages to maintain high levels of robustness for the following reasons: Query forwarding is a probabilistic process, meaning that nodes with the largest values do not get necessarily chosen. This is important, as object locations may change frequently and wrong next-hop choices may occur. No neighbors are excluded because of a low probability and node failures cannot interfere with the algorithm's normal operation (unless a requester's neighbors are fewer than k , in which case all of them are chosen). Furthermore, no more traffic for index maintenance is placed on the network. Our algorithm also utilizes its *unlearning* feature, which enables walkers to be redirected if previously recovered objects are missing. Finally, the probability of query failure is greatly reduced with the use of a large number of

walkers. The changes in topology or object locations must simultaneously affect all successful paths in order for a miss to occur. The metric we expect to be reasonably affected is the number of hits per search, as some paths to discovered objects frequently "disappear".

We can see that *APS* achieves very high success rates even with few deployed walkers. It is steadily around 40% more accurate than *Random Walks*. We also noticed that *w-APS* showed a small increase of around 4% in successful searches over our standard method for the static environment. As predicted above, the accuracy is not greatly influenced by node departures. For the less dynamic run, the amount of decrease is almost zero, while it remains within only 5% for relatively large ($k \geq 8$) values.

One would expect that our method produces a much larger number of messages compared to *Random Walks* due to the update process, but this is not the case: The majority of walkers in *APS* are successful and only few of them reach *TTL* hops away. In *Random Walks*, about 70% of the walkers fail and waste *TTL* messages each. To a lesser extent, objects are equally discovered at all possible distances in the random method, while our scheme discovers more objects closer to the requesters. The results confirm our case: The difference in messages is about 15 for the *pessimistic* approach, which proves that a single update policy is not suitable for all ranges of requests. The *s-APS* improvement has the same very low production as the random algorithm. This effect is enhanced if we recall that no message exchange is necessary for peer join/leave/update operations. Only in the highly dynamic setting do we see an increase in the average production, which is at most 5–7 messages. This gap appears because of the frequent broken paths to objects, causing walkers to travel more inside the network.

The last graph in Figure 2 depicts the average number of discovered objects per query. *APS* puts the walkers to a much better use, discovering around 4 times as many objects as its competitor. This is extremely important for current popular P2P applications, giving the user a much broader choice for download. This characteristic comes as a result of its high success rate and very few walker collisions

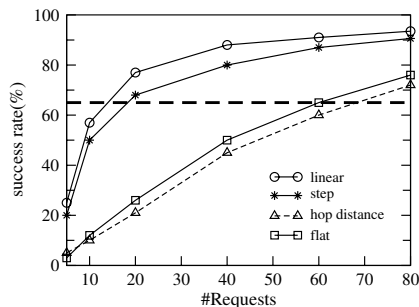


Figure 3. Comparing the accuracy of different index update functions

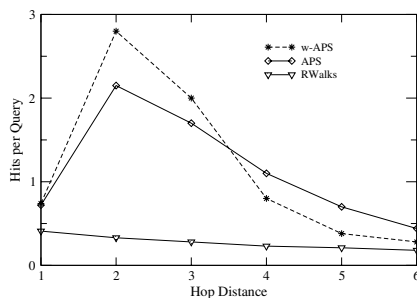


Figure 4. Hits per query vs. hop distance

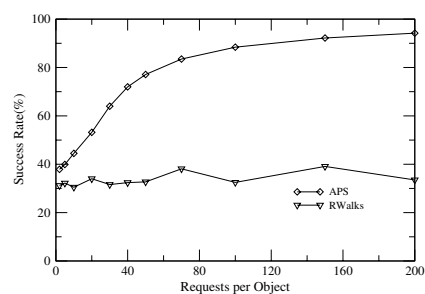


Figure 5. Accuracy vs. requests per object

(our results showed a reduction of 1 to 2 orders of magnitude in duplicate messages compared to *Random Walks*). In the dynamic settings, the maximum reduction in the number of hits is around 25% and 40% for the less dynamic and more dynamic runs respectively. These numbers occur for large values of k , where the probability of node departures affecting the walkers increases.

Figure 4 shows how the hits are distributed over their distance from the requesters, for the default parameters (static setting). While *Random Walks* discovers about the same amount of objects throughout the 1 to TTL range, *APS* is more biased into the first half of this range. The *w-APS* technique significantly improves on this characteristic, trading distant objects (hops 4 to 6) with closer ones (hops 1 to 3). The majority of discovered objects are now found with few messages, while fewer objects are discovered with more messages. The results for dynamic settings are similar, the only difference being the reduction in hits we saw before. We also noticed that our algorithm becomes more biased into discovering nearby objects as the number of replicas inside the network increases. This happens because the walkers have a broader selection of paths to objects and can, therefore, choose the shortest.

Figure 5 displays how the number of requests affects accuracy. With just 1% object replication ratio, $k = 10$ and $TTL = 5$, we varied the number of requests per object using a uniform distribution for both storage and requests on the default graph. We can see that the accuracy of our method improves significantly with only a small increase in requests. At the same time, *Random Walks* is steadily below 40%, regardless of the number of requests.

4.4. Results for more environments

In this section, we compare *s-APS* with *Random Walks* over four different graphs: The default one, a 10,000-node random graph with $d = 4$ (similar to Gnutella-type graphs), a 50,000-node random graph with $d = 10$ and a 10,000-node power-law (PLAW) graph with $d = 4.4$. Table 1 presents the two algorithms' performance in the highly dynamic setting

with the respective results from the static runs in parentheses.

First, we test the methods using a uniform distribution for both requests and storage in the default graph. The replication ratio for each object is set to 1% and each of them receives 30 queries by each requester node. We clearly notice that *s-APS* greatly benefits from such a setup, delivering over 94% in success rate (a mere 2% decrease from the static run) and discovering more than 10 times more objects than *Random Walks*.

On a similar graph with smaller out-degree and $k = 5$, *s-APS* is still 40–50% more accurate, 5 times more effective in locating objects and almost as bandwidth-efficient as the random method. The results are worse compared to the default graph because of the smaller out-degree and fewer walkers used.

Our simulations on the 50,000-node random graph justify our prediction that the graph size cannot influence the performance of *APS*. The results were a little worse from the ones in the original graph, because the quality of the new graph was worse (many more disconnected components were present). We notice the success rate is about 8% lower from the static case, while the number of discovered objects is almost halved.

Our results on the 10,000-node power-law graph show an even greater gap in the performance of the two algorithms. Our method delivers about 4 times more results and exhibits a success rate three times bigger than *Random Walks*'. The success rate for *s-APS* drops by around 9% and discovered objects decrease by 37%, while message production slightly decreases.

In these simulations, our method kept its message production at the same levels with the static runs, wasting at most 5 extra messages per search, a direct proof that it does not impose more burden on network traffic. As expected, the success rate shows only a small decrease, from 2% to 12%. *APS* exhibits remarkable robustness even in such fast-changing environments. These results also show that our method maintains its relative performance gains over the

Table 1. Results for more environments

Graph-Distr.	<i>s-APS</i>			<i>Random Walks</i>		
	<i>Succ%</i>	Mesg	Hits	<i>Succ%</i>	Mesg	Hits
<i>10K-Rand</i> (<i>d=10,Unif</i>)	94.1 (96.1)	58.5 (53.5)	4.3 (7.2)	32.3 (38.2)	41.8 (49.6)	0.4 (0.5)
<i>10K-Rand</i> (<i>d=4,Zipf</i>)	70 (82.2)	17.3 (18.2)	1.4 (2.25)	26.0 (34.5)	12.0 (15.0)	0.3 (0.5)
<i>50K-Rand</i> (<i>d=10,Zipf</i>)	79.3 (87.6)	48.4 (47.0)	2.4 (5.7)	55.6 (57.6)	39.5 (45.7)	1.3 (1.4)
<i>10K-PLAW</i> (<i>d=4.4,Zipf</i>)	67.6 (76.1)	13.0 (14.9)	1.11 (1.76)	21.0 (31.6)	9.0 (12.0)	0.3 (0.5)

different environments.

4.5. Comparison with GUESS

Lastly, we present results comparing *s-APS* with an implementation of *GUESS* [4] on a random *hybrid* graph with 6500 peers, 500 of them being super-peers. Each super-peer is connected to 12 leaf-nodes on average. Links exist only between super-peers and between an super-peer and its leaf-nodes. In our *GUESS* implementation, initiating super-peers forward queries to *k* randomly chosen neighbor super-peers. Query and replication distributions are set to their default values. Since it is impossible to directly compare the two methods for the same *k* and *TTL* values, we select simulations where the two algorithms had similar performance in one of two important metrics: Messages and hits per query. The results are presented in Table 2 and the comparison metric is typed in boldface. For similar message consumption, our scheme exhibits higher success rates and delivers 4 to 5 times more results. For similar hits per search, our scheme produces 4 to 5 times fewer messages and always outperforms *GUESS* in accuracy. *APS* achieves these results taking no advantage of the hybrid topology that *GUESS* utilizes.

5. Conclusions

This paper presents *APS*, an adaptive search technique for unstructured P2P networks. *APS* deploys probabilistically directed walkers by utilizing information from past searches. This allows for fast, joint learning, while being extremely bandwidth-efficient. Peers are required to keep indices only relative to their neighbors, while no message exchange is necessary for any dynamic network event, local or global. Our simulations on a variety of environments demonstrated the versatility of the proposed technique. Results show that *APS* achieves high performance being almost as bandwidth-efficient as *Random Walks*. It discovers 4 times as many objects and delivers very high success rates compared to the *Random Walks* and *GUESS* methods. Finally, we demonstrated our algorithm's ability to maintain

Table 2. Comparison with GUESS

Metric	<i>s-APS</i>			<i>GUESS</i>		
	<i>Succ%</i>	Mesg	Hits	<i>Succ%</i>	Mesg	Hits
<i>Messages</i>	97.7	16.3	5.22	63.9	16.1	1.28
	98.6	22.0	7.01	65.6	22.2	1.87
	99.7	33.2	11.39	84.0	33.1	2.55
<i>Hits</i>	81.0	3.2	1.33	63.9	16.1	1.28
	94.6	8.7	3.42	86.4	45.0	3.70
	97.9	16.5	5.42	94.5	65.1	5.60

these features even in rapidly changing environments, exhibiting a high degree of robustness.

References

- [1] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. *In press, Phys. Rev. E*, 2001.
- [2] J. Chu, K. Labonte, and B. Levine. Availability and locality measurements of peer-to-peer file systems. In *SPIE*, 2002.
- [3] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, 2002.
- [4] S. Daswani and A. Fisk. Gnutella UDP extension for scalable searches (*GUESS*) v0.1.
- [5] D. Tsoumakos and N. Roussopoulos. A Comparison of Peer-to-Peer Search Methods. In *WebDB*, 2003.
- [6] Gnutella website: <http://gnutella.wego.com>.
- [7] C. Jin, Q. Chen, and S. Jamin. Inet: Internet Topology Generator. Technical Report CSE-TR443-00, Department of EECS, University of Michigan, 2000.
- [8] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *CIKM*, 2002.
- [9] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, 2002.
- [10] D. Menascé and L. Kanchanapalli. Probabilistic Scalable P2P Resource Location Services. *SIGMETRICS Perf. Eval. Review*, 2002.
- [11] Napster website: <http://www.napster.com>.
- [12] .NET home page: <http://www.microsoft.com/net/>.
- [13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *SIGCOMM*, 2001.
- [14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2001.
- [15] The impact of file sharing on service provider networks. An Industry White Paper, Sandvine Inc.
- [16] SETI@home web site: <http://setiathome.ssl.berkeley.edu/>.
- [17] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM*, 2001.
- [18] M. Stokes. Gnutella2 specifications part one.
- [19] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, 2002.
- [20] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM*, 1996.