

# A Two-Level Random Walk Search Protocol for Peer-to-Peer Networks

Imad Jawhar and Jie Wu  
 Department of Computer Science and Engineering  
 Florida Atlantic University  
 Boca Raton, FL 33431

**Abstract**—Peer-to-peer systems are finding more popularity as a means of sharing large volumes of data between different users. Each node contains a certain amount of data which is typically files but can also be of other types such as records in a relational database. There have been different search algorithms that were developed which enable a certain node to find data that is distributed among the other nodes in the network [2][3]. Each of these algorithms have certain advantages and disadvantages. This paper introduces a search protocol to be used in peer-to-peer networks. It uses a two-level random walk strategy. The querying node selects  $k_1$  random walks with  $TTL_1(\text{time-to-live})=l_1$ . When the  $TTL_1$  timer expires at a particular node, each thread will then generate  $k_2$  threads which will perform  $k_2$  random walks from that node with  $TTL_2=l_2$ . This protocol has the advantage of increasing the coverage area with the same number of messages and reducing the redundancies which take place when peer nodes repeatedly receive the same query message. On the other hand, the price paid by this strategy is an increase in the average length of the discovered path.

**Keywords:** Distributed Information Retrieval, Peer-to-Peer, Search, Unstructured.

## I. INTRODUCTION

As the Internet and its applications continue to grow in such rapid fashion, peer-to-peer (P2P) systems allow users to share large volumes of data which is distributed among a vast amount of nodes [2][3][6]. Each of these nodes can play the roles of both client and server. At the core of the effectiveness and usability of peer-to-peer systems is the adoption of efficient search algorithms with a low level of redundant hits in the same nodes (it is desirable for nodes to only be searched once), small message overhead, and good scalability. Currently, there are several architectures for P2P systems [3]:

- Centralized systems: such as Napster. In these systems a centralized server keeps track of a directory which is continuously updated and maintained. Such centralized architectures have the disadvantages of a centralized point of failure, low scalability and are relatively expensive.
- Decentralized but Structured: These are "loosely structured" systems in which the P2P overlay (connections among the P2P nodes) is closely controlled and files are placed in specific non-random locations which are intended to make queries more efficient. Freenet is an example of such P2P system [4]. These systems are not

as available in practice, even though a good amount of research about them has been conducted in the literature. Such designs might not be well suited for transient populations which characterize current P2P environments.

- Decentralized and unstructured: such as Gnutella [5]. These systems do not have a particular structure or centralized directory and there is no precise control over the network topology or file placement. Nodes are joined in the network following some loose rules [1]. In order to find a particular file, a node broadcast queries to its neighbors. The search algorithm used are based on flooding with a certain radius (indicated by a specified TTL in number of hops). Such architecture is well suited for the current P2P environment where nodes regularly join and leave the system.

In this paper, we focus on the decentralized and unstructured architecture. We present an efficient search algorithm which increases the total number of nodes searched for a certain total number of search step, and reduces the redundancy or average number of times a particular node is searched. The remainder of the paper is organized as follows. Section 2 discusses related work that has been done in this field. In section 3, we present our protocol. In section 4, we present and discuss the simulation results. In section 5, we present some enhancements which can be applied to extend this protocol. The last section will present conclusions and future research.

## II. RELATED WORK

In [3], an algorithm is presented which is more efficient than the flood-based search. It referred to in this paper as a one-level  $k$ -walk algorithm (uses  $k$  random search threads) and it is intended to reduce the load generated by each search query. The algorithm works in the following manner. A node that is searching for a particular file, initiates a query with a TTL timer =  $l$  hops. It then broadcasts it to  $k$  randomly selected neighbors. Each node that receives the query processes it and determines if it has the desired information (file in this case). If that is the case it responds to the source. Otherwise, if it does not have the required information, it checks the TTL value, if it is more than zero it decrements it by one and forwards the query to one randomly selected neighbor. When the TTL value reaches zero the query is not forwarded any further.

Our proposed algorithm uses a two-level random walk. It works in the following manner. The source node creates a query with  $TTL_1=l_1$ . It then broadcasts the query to

$k_1$  randomly selected neighbors (the value for  $k_1$  here is much smaller than that of the one-level  $k$ -walk algorithm,  $k$ ), initiating  $k_1$  search threads. When a node receives the message, it checks the TTL1 value. If it is more than zero, the node decrements the TTL1 timer, and randomly selects one neighbor to which it forwards the query. If the TTL1 value is zero, then this node is considered an *edge node* and it "explodes" the query into  $k_2$  queries by randomly selecting  $k_2$  neighbors and forwarding the query to each of them. Each of the  $k_2$  new threads is given a TTL2 value of  $l_2$ .

As demonstrated by the simulation, our algorithm reduces redundancy by decreasing the average number of times a node is searched. In the one-level  $k$ -walk algorithm  $k$  random threads are generated from the source and they are likely to have "thread collisions" (i.e. threads run into each other) especially near the source. This results in having redundant hits in the same nodes (nodes being searched multiple times). On the other hand, the proposed two-level algorithm sends fewer threads from the source node which result in a smaller probability of thread collisions near the source. Each of the  $k_1$  threads will then explode into  $k_2$  threads once it is "sufficiently" away from the source and the other threads. This way, the same number of search threads can be generated ( $k=k_1*k_2$ ) but with a larger number of nodes searched and a smaller probability of redundant searches to the same nodes using the same number of total search steps.

### III. OUR PROTOCOL

#### A. Protocol Overview

Before presenting an overview of our algorithm, it would be useful to define what is meant by the term "Query Processing". Processing a query means the following. When a node receives a query message, it may forward the query to a set of its neighbors, depending on the routing policy. The node will also process the query over its local collection and produce results (such as pointer to files). If any results are found at that node, the node will send a single response message back to the query source via the reverse path travelled by the query. The total result set for a query is the bag union of results from every node that processes it. [6].

Our algorithm works in the following manner. When a node wishes to send a query with a certain search key, it composes a search message and broadcasts it to  $k_1$  randomly selected neighbors. The message has an initial TTL1 =  $l_1$  hops. When an intermediate node receives this message, it checks the TTL1 timer. If the latter is still more than 0 then it decrements the timer by one, selects one random neighbor and forwards the message to it. This process continues until one of the nodes, say node  $E$ , receives the message with an expired TTL1 timer (i.e. TTL1 = 0). We call such a node an *edge node*. The message will then "explode" into  $k_2$  search messages forwarded from this node. Specifically, node  $E$  will compose a message with TTL1=0, and a second timer TTL2= $l_2$ . It will then randomly select  $k_2$  of its neighbors, excluding the one it just received the message from, and broadcast the message to them. Figure 1 shows an example illustrating this process. At level one, a source node sends  $k_1$  random messages

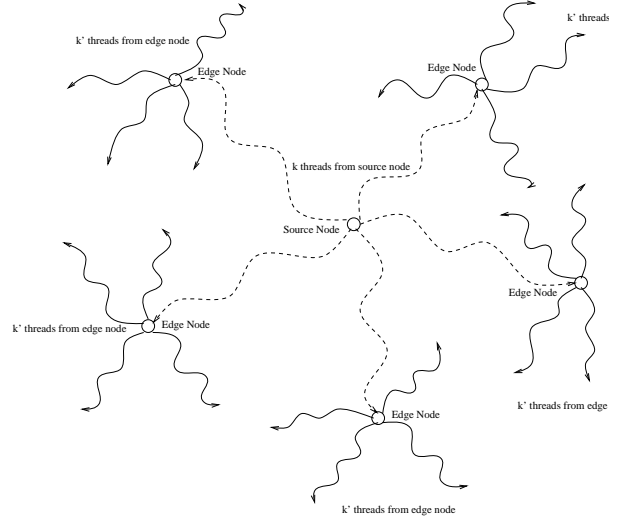


Fig. 1. Two level random walk. At the first level the source sends  $k_1$  random threads, and each thread explodes to  $k_2$  threads when TTL1 expires (at the  $k_1$  edge nodes)

to a set of  $k_1$  randomly selected nodes of its neighbors. This constitutes  $k_1$  threads (or random walks) which travel from the source node to the edge nodes (a node where TTL1 expires). Each of the  $k_1$  threads will then explode into  $k_2$  threads (with TTL2 =  $l_2$ ) at each of the  $k_1$  edge nodes.

As demonstrated by the simulation, conducting the search in this fashion has the advantage of low probability of redundant hits and a larger number of searched nodes. This is the case since the first  $k_1$  threads will only explode into  $k_2$  threads after travelling a TTL1 =  $l_1$  distance from the initial node, and each other, and therefore the message has a higher probability of "hitting" new nodes and not running into other threads which cause redundant searches of the same nodes. On the other hand, this efficiency comes at the price of longer average length for the discovered path. For many application, however, this would an acceptable compromise.

#### B. Algorithm at an intermediate node $y$

The algorithm executed by an intermediate node that receives a search message is presented in Algorithm 1. It works in the following manner. When a node  $y$  receives the message from another node  $x$ , if it already received the message, identified by the source/searchKey/sessionID tuple, it drops it. Otherwise, it processes the query. It then checks to see if it is an intermediate node at level 1 by checking if TTL1 (that is in the message) is more than 0. If it is, then the node decrements TTL1 by 1 and forwards the message to a randomly selected neighbor  $z$  that is not  $x$  (the node it just received the message from). Otherwise (TTL1 = 0),  $y$  checks to see if it is an *edge node* by determining if TTL2 that is included in the message is equal to  $l_2$ . If that is the case, then  $y$  is an *edge node*. So, it decrements TTL2 by 1 and explodes the current first level thread into  $k_2$  second level threads by forwarding the message to  $k_2$  randomly selected neighbors other than  $x$ . If however TTL2 is not equal to  $l_2$ , then  $y$  checks if TTL2 is more than 0 to determine if it is an intermediate node in the second level.

---

**Algorithm 1** The algorithm at an intermediate node
 

---

```

When a node  $y$  receives a search message from a node  $x$ 
if  $y$  received this message identified by
 $S/searchKey/sessionID$  before then
  drop this message and exit this routine
else
  process query.
  if  $TTL1 > 0$  then
    Decrement  $TTL1$  by 1
    Forward message to one randomly selected neighbor  $z$  of  $y$ 
    other than  $x$ .
  else
    if  $TTL2 = l_2$  then
      decrement  $TTL2$  by 1
      forward message to max of  $k_2$  neighbors not including  $x$ .
    else
      if  $TTL2 > 0$  then
        decrement  $TTL2$  by 1.
        forward message to one randomly selected neighbor  $z$ 
        of  $y$  other than  $x$ .
      end if
    end if
  end if
end if
  
```

---

If that is the case, then it decrements TTL2 by 1 and forwards the message to a randomly selected neighbor  $z$  other than  $x$ . If on the other hand TTL2 is equal to 0, then  $y$  is the last node in the second level. Consequently, having processed the query already,  $y$  drops the search message which in effect would not be forwarded any further and this would be the end of a second level thread.

### C. Analysis

When the algorithm described earlier is compared with the one-level k-walk algorithm described in [3], the following points can be stated:

The total number of steps taken by the one-level k-walk is:  $k * l$ , where  $k$  is the number of threads and  $l$  is the number of steps for each thread.

The total number of steps taken by the two-level random walk is:  $k_1 * l_1 + k_1 * k_2 * l_2$ .

The idea behind the two-level k-walk algorithm is that we try to minimize the redundancy of hits (searching nodes more than once) near the source that would be experienced by the one-level k-walk algorithm. As mentioned earlier, this is achieved by having a smaller number of threads leave the source and wait a number of steps,  $l_1$ , before multiplying into more threads. Less redundancy of hits would be achieved because the larger amount of search threads would take place when these threads are sufficiently away from the common source and therefore each other. This would be particularly effective and desirable if the source had a smaller node degree (few neighbors) than the amount of  $k$  threads that we would like to generate in the one-level k-walk case. This is because in such case, a forced amount of redundant hits would take place near the source because multiple threads would have to go through the same neighbors. This would happen because the amount of threads that need to be dispatched from the

source is larger than the number of neighbors. This inefficiency does not exist in our algorithm, because we have a relatively small amount of threads which initially leave the source. Even though, we still end up with the same amount of threads as the one-level algorithm as the search progresses to the outer reaches of the network in the second level.

## IV. SIMULATION

We performed a simulation in order to study the performance of our algorithm and compare it to the one-level k-walk algorithm [3]. In the simulation, a connected random graph was generated which contained a number of nodes (250 in this experiment). In order to study the performance of the algorithms under different conditions, several simulation parameters were varied. However, in all cases, the total number of steps taken by either algorithm was the same. We placed copies in a percentage of the nodes. In our simulation, we used 30 percent. However, without loss of generality, the number could be more or less depending on the application involved. Also the density of the generated graph was varied by using average node degrees of 5, 10, and 25. We used these values in order to focus on the performance of the search algorithm with different graph densities without considering particular applications. We considered three values for the ratio of number of threads to the node degree in the one-level k-walk algorithm. The ratios used are: 0.3, 1, 2. The number of first and second level threads,  $k_1$  and  $k_2$  respectively, was derived by considering the number of threads in the one-level k-walk  $k = k_1 * k_2$ . Assuming that  $k_1 = k_2$ , we derive  $k_1 = k_2 = \sqrt{k}$ . On the other hand, three values for the total number of steps,  $N$  (which is kept the same for both algorithms), were used: 50, 100, and 150 steps. From the total number of steps and the number of threads used we derived the value for the length of each thread in the one-level k-walk,  $l = N/k$ . Then, we can calculate the values of the parameters for the two-level k-walk algorithm. Since,  $N = k_1 * l_1 + k_2 * k_1 * l_2$ , and we set  $k_1 = k_2$  and  $l_1 = l_2$  (for simplicity), we derive  $l_1$  and  $l_2$  using the following formula:  $l_1 = l_2 = N / (k_1 + k_1^2)$ .

The tabulated simulation results are shown in Figure 2. Figure 3 shows the charts for the number of copies found (top) and the number of redundant hits (bottom) for each of the two algorithms and for graphs with average node degree of 5. Note that, in the chart, the number of copies found and number of redundant hits are shown for different values of the total number of steps  $N$ . Figure 4, and Figure 5 show the charts for graphs of average node degrees of 10 and 25 respectively. In general, the simulation shows a considerable advantage of the two-level k-walk algorithm in the number of redundant hits. This advantage is largest with higher density graphs, and as the number of threads leaving the source increases. The number of copies found by the two-level k-walk algorithm is also higher, although to a lesser degree, under the same conditions. Under these conditions, in the one-level k-walk algorithm case, as the number of threads increases, different threads have to go through the same neighbors of the source. This results in decreased search efficiency since we are wasting search steps by checking the same nodes. This problem is not

Number of nodes =250  
copies = 30%

Copies found			
node degree = 5			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	13.30	13.07	11.60
Two level walk, N=50	12.93	13.23	13.17
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	17.90	19.53	15.67
Two level walk, N=75	18.63	19.10	18.10
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	23.83	23.43	22.57
Two level walk, N=100	24.50	23.77	23.57
node degree = 10			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	12.70	12.20	7.93
Two level walk, N=50	13.17	12.83	10.07
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	19.70	17.47	13.23
Two level walk, N=75	19.17	18.93	15.33
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	24.87	24.10	22.03
Two level walk, N=100	25.20	24.20	24.13
node degree = 25			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	13.00	12.90	7.47
Two level walk, N=50	12.10	13.70	13.83
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	19.63	19.17	20.43
Two level walk, N=75	19.20	18.97	24.13
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	23.60	24.60	19.87
Two level walk, N=100	22.80	24.57	25.77

Redundant hits			
node degree = 5			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	5.27	6.07	10.83
Two level walk, N=50	4.93	5.33	6.53
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	11.33	10.30	16.23
Two level walk, N=75	11.30	10.40	11.27
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	19.47	19.23	24.10
Two level walk, N=100	18.57	20.47	23.33
node degree = 10			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	4.17	6.00	13.80
Two level walk, N=50	4.03	4.87	2.67
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	9.93	9.93	17.60
Two level walk, N=75	10.47	8.60	6.63
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	17.87	18.83	26.33
Two level walk, N=100	16.57	18.77	17.47
node degree = 25			
N=50			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=50	3.83	4.63	24.30
Two level walk, N=50	4.90	3.77	4.73
N=75			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=75	8.37	10.97	34.23
Two level walk, N=75	9.53	8.37	17.33
N=100			
	NT=0.3*ND	NT=1*ND	NT=2*ND
One level walk, N=100	16.27	17.80	34.10
Two level walk, N=100	16.50	15.47	16.07

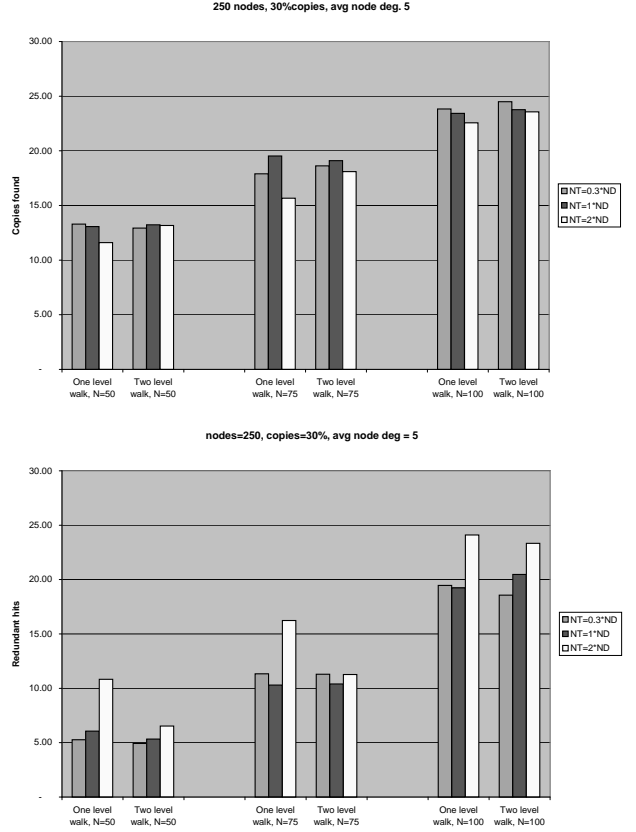


Fig. 2. Number of copies found (left) and number of redundant hits (right) for one-level and two-level k-walk algorithms as the total number of steps,  $N$ , is varied - for networks with an average node degrees of 5, 10 and 25. Note: ND: Node Degree, NT: Number of Threads

Fig. 3. Number of copies found (top) and number of redundant hits (bottom) for one-level and two-level k-walk algorithms as the total number of steps,  $N$ , is varied - for networks with an average node degrees of 5. Note: ND: Node Degree, NT: Number of Threads

encountered in our algorithm since the source needs to send a smaller number of threads in the first level that will later explode into more threads after they get farther away from the source, and each other, and have a lower probability of thread collisions which causes those threads to redundantly search the same nodes. This explains why the two-level k-walk algorithm consistently have a lower number of redundant hits. Under these circumstances, and as expected, the simulation shows that our algorithm, on average, finds more copies for the same number of total steps. However, this comes at a price of added latency since the average path from the source to the target is expected to be relatively longer. This is the case because the search thread travels away from the source first before exploding into more threads.

## V. ALGORITHM ENHANCEMENTS

It is logical to expect that multiple copies of the same type would tend to be clustered in the same neighborhoods of a graph. For example if a French song exists in a certain geographic area, it is more likely that similar songs exist in that same area. This is due to the principle of the locality of reference that would be logically expected to be exhibited by search objects such as music, video, and other files that users might be seeking. In this situation, a variation of our algorithm

can be deployed which would be adapted to such conditions. The source sends a number of threads at the first level, and each of the threads explodes to a predetermined number of second level threads as soon as a copy is found. Such a strategy would be effective in these described circumstances because the "finding of a copy" would cause us to believe that many more similar copies are more likely to be clustered in the same area, and this would increase the probability of finding copies using the same number of steps. A variation of this algorithm is to have a "probability of exploding" for each thread which would be increased each time a new copy is found. Another variation which can be considered is to have multiple levels of search. At each level a thread explodes into multiple threads. Many parameters can then be varied and the performance can be analyzed to provide the best set of parameters for different network characteristics.

## VI. CONCLUSIONS AND FUTURE RESEARCH

We presented an two-level k-walk protocol for conducting search in a network. The protocol can be used in different applications such as decentralized and unstructured peer-to-peer networks. It uses a two-level random walk, where the querying node broadcasts its query message to  $k_1$  randomly selected neighbors with a  $TTL1 = l_1$ . Each node will then

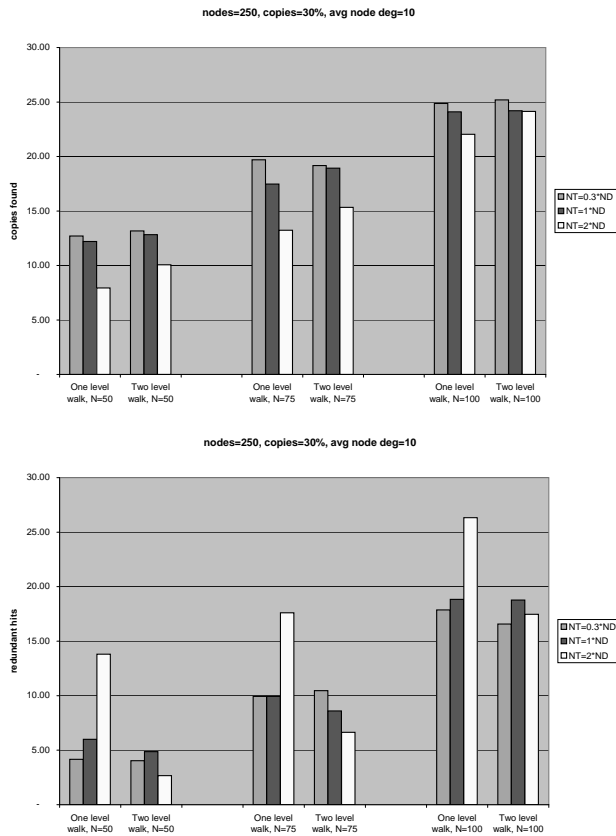


Fig. 4. Number of copies found (top) and number of redundant hits (bottom) for one-level and two-level k-walk algorithms as the total number of steps,  $N$ , is varied - for networks with an average node degrees of 10. Note: ND: Node Degree, NT: Number of Threads

decrement TTL1 and forwards the message to one randomly selected neighbor. When the TTL1 expires, the corresponding node (named *edge node*) will "explode" the thread into  $k_2$  search threads with  $TTL2 = l_2$ . On average, this algorithm has an advantage of lower redundancy of hits (multiple searches of the same nodes), and therefore higher efficiency, over the one-level k-walk algorithm presented in the literature. However, this advantage comes at the cost of relatively longer average length of the discovered path. Certain optimization techniques can be applied to the proposed algorithm, and more simulations can be conducted in order to further analyze and improve its efficiency, and increase its adaptability to different networking environments.

## REFERENCES

- [1] Clip2.com. The gnutella protocol specification v0.4. In [http://www.limewire.com/developer/gnutella\\_protocol\\_0.4.pdf](http://www.limewire.com/developer/gnutella_protocol_0.4.pdf), 2000.
- [2] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. *Proceedings of the eleventh international conference on Information and knowledge management, McLean, Virginia, USA*, pages 300–207, 2002.
- [3] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. *ICS'02, New York, USA*, June 2002.
- [4] Freenet website. <http://freenet.sourceforge.net>.
- [5] Gnutella website. <http://gnutella.wego.com>.

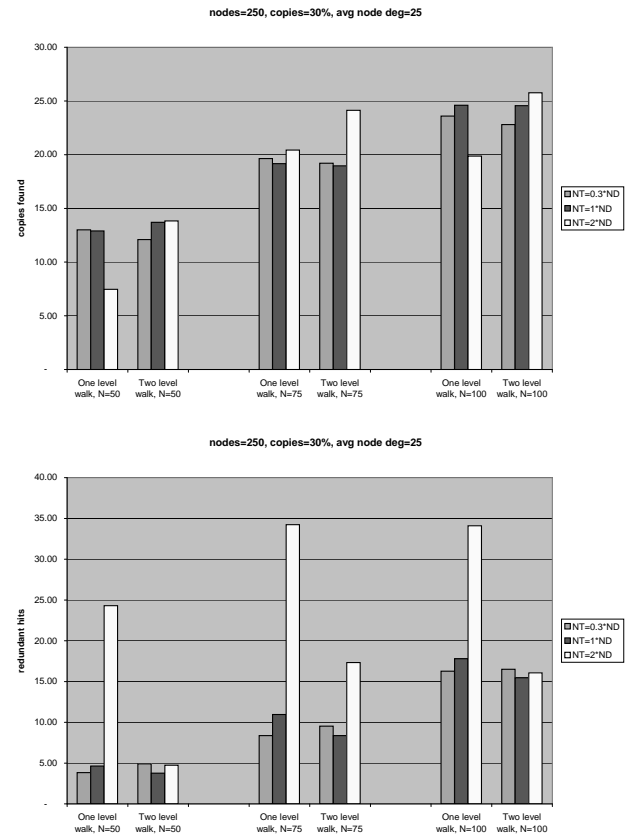


Fig. 5. Number of copies found (top) and number of redundant hits (bottom) for one-level and two-level k-walk algorithms as the total number of steps,  $N$ , is varied - for networks with an average node degrees of 25.

- [6] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 5–14, July 2002.