ENSC 835: HIGH-PERFORMANCE NETWORKS

Simulating Various Search Strategies for Gnutella

FALL 2003

Final Project

Chan, Chun Wai (John)

Simon Fraser University

cchany@sfu.ca

http://www.sfu.ca/~cchany/ENSC835

TABLE OF CONTENTS

Page

Table of Contents ii

List of Tables iii

List of Figures iv

Abstract v

I INTRODUCTION 1

II CURRENT METHOD FOR SEARCHING - BFS 2

III PROPOSED METHODS FOR SEARCHING 3

A. The Randomized BFS 3

B. The k-Walker Random Walk 3

IV IMPLEMENTATION DETAILS 4

A. Simulation Core 4

1. Implementation of the Search Strategies in NS 5

2. TCP Connection Management and Message Passing 6

3. Data Collection 7

4. Data Structures 7

B. Visualizing the Simulations 8

C. Topologies, Files and Search Requests Generations 9

D. Simulation Scenarios and Helper bash Scripts 10

E. Statistics Helpers 11

V RESULTS AND DISCUSSION 12

VI CONCLUSIONS 21

VII FUTURE WORK 22

VIII REFERENCES 23

APPENDIX 1: CODE LISTING 24

LIST OF TABLES

Page

Table I. Search request packet makeup..... 11

Table II. Simulation scenarios used for discussion..... 12

Table III. Simulation results for scenario A..... 13

Table IV. Simulation results for scenario B..... 13

Table V. Simulation results for scenario C..... 16

Table VI. Simulation results for scenario D. 16

Table VII. Simulation results for scenario E. 19

Table VIII. Simulation results for scenario F. 19

LIST OF FIGURES

Page

Figure 1. BFS. 2

Figure 2. Randomized BFS with preset percentage as 50%. 3

Figure 3. Randomized BFS with k set to 2. 4

Figure 4. Class hierarchy for the search classes. 5

Figure 5. Class relationships for TCP connection management. 6

Figure 6. Amount of network traffic over time for scenario A. 14

Figure 7. Amount of network traffic over time for scenario B. 15

Figure 8. Amount of network traffic over time for scenario C. 17

Figure 9. Amount of network traffic over time for scenario D. 18

Figure 10. Amount of network traffic over time for scenario E. 20

Figure 11. Amount of network traffic over time for scenario F. 21

ABSTRACT

Gnutella is a decentralized data-sharing P2P network protocol. A host initiating a search in the network will send its search request to all its neighbours. Upon receiving a search request, a node will forward the request to all its neighbours. This process continues until a pre-specified radius has been reached. This search method poses a heavy burden on the network because the amount of network traffic grows exponentially. Various studies have suggested alternative approaches for searching in the Gnutella architecture. This paper will compare three different search techniques, including BFS, Randomized BFS and k-Walker Random Walk, by capturing the amounts of network traffic induced by simulating them in NS simulator.

I INTRODUCTION

In a decentralized peer-to-peer (P2P) network, individual computers of "equal roles and capabilities exchange information and services directly with each other" [5]. Due to the widespread of the Internet and increasing demand for limited computing resources such as disk space and bandwidth, data-sharing P2P systems become a popular way for people to share huge amount of data. Because of the large number of users in the P2P network, simultaneous searches induce large amount of network traffic. Therefore, the ability to search for the desired data without imposing a large burden on the network bandwidth is one of the most wanted and important features in data-sharing P2P systems.

Gnutella is one of the most well known decentralized P2P data-sharing protocols. According to Gnutella Protocol Development [6], the current stable version of Gnutella is Gnutella 0.4. The Gnutella Protocol Specification v0.4 [1] uses a broadcasting strategy to perform searches. It has been shown that it is not scalable because of the large amount of network traffic generated by the searches [4]. Therefore, many literatures have proposed alternative search techniques to be deployed on the existing Gnutella architecture [2, 3]. Although the literatures have done performance analysis for their own search techniques, the results cannot be compared because the experimental conditions varied wildly.

The purpose of this project is to investigate and compare three different search techniques including the Breadth First Search (BFS), the Randomized BFS, and the k-Walker Random Walk, in the Gnutella network in a uniformed simulation environment. NS version 2 simulator is used to perform the simulations. The different search strategies will be compared in terms of the amount of network traffic generated and the success rate of a search.

The next section of this paper describes the current search technique, the BFS, of the Gnutella network. The third section introduces other techniques including the Randomized BFS and k-Walker Random Walk, proposed by the literatures. Section four outlines the design and implementation details for the simulation. Finally, section five describes the simulation results, findings and gives future directions for the research.

II CURRENT METHOD FOR SEARCHING - BFS

In Gnutella Protocol v0.4, each computer keeps an index of its own data. When a computer starts a search, the request is sent to all computers that the requesting computer has direct connection with, also known as neighbouring computers. Upon receiving a search request, a computer will check if it has the data being requested and send a response to the requesting computer if the data is available. After that the computer will forward the request to all its neighbouring computers. The process continues until a preset limit on the number of forwarding (also known as TTL) is reached. If a node has received a duplicate search request, the computer will ignore that request. This search technique is known as BFS. This mechanism eliminates single-point-of-failure completely and guarantees to find the needed data in a preset area in the network if it is present. However, the bandwidth requirement for this method is too high and thus it cannot scale well [4].

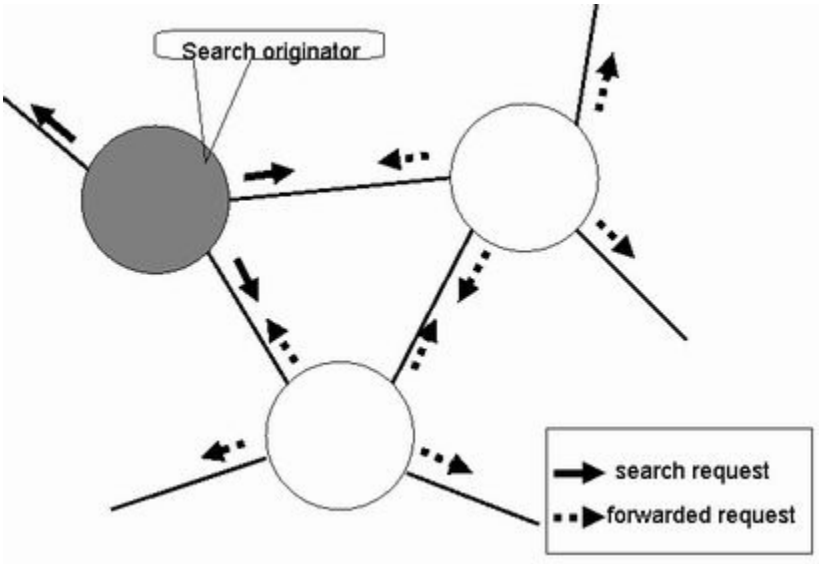

Figure 1. BFS.

III PROPOSED METHODS FOR SEARCHING

Recent literatures have suggested various improvements to the existing Gnutella search mechanism. Kalogeraki, et al. [2] has suggested the Modified Random BFS. Lv, et al. [3] has suggested the k-Walker Random Walk.

A. The Randomized BFS

The Randomized BFS (known as Modified Random BFS [2]) is similar to Gnutella's BFS. The only difference is that instead of forwarding the request to all neighbouring computers, the request is only forwarded to a preset percentage of neighbouring computers chosen randomly. Laboratory experiments show that this method reduce the bandwidth cost by almost 80% at the expense of retrieving half of the data available as compared to BFS [2]. The experiment is based on a computer simulation of a network of 100 computers having a fixed network organization.

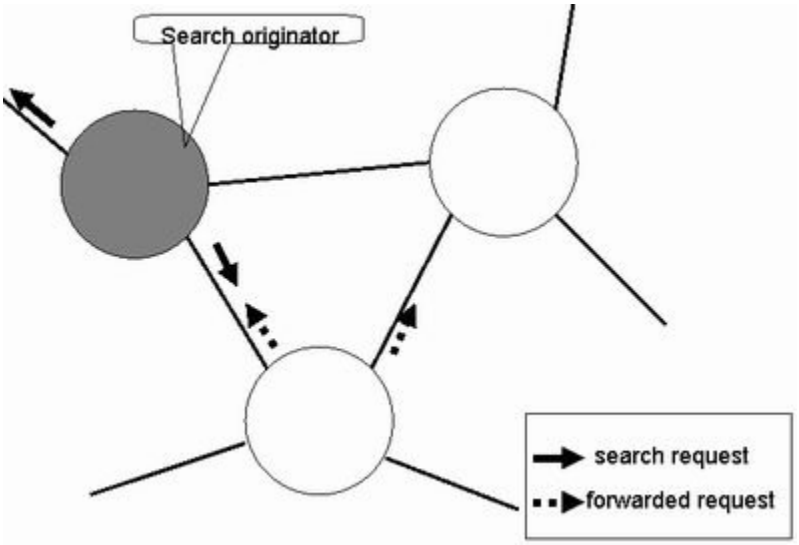

Figure 2. Randomized BFS with preset percentage as 50%.

B. The k-Walker Random Walk

Lv, et al. [3] introduces the k-Walker Random Walk technique. A requesting computer sends search requests to k neighbouring computers chosen randomly where k is a preset constant value. Upon receiving a search request, the computer checks if it has the

data that is being requested for. If the data exists, a response is sent to the requesting computer and the search ends. If not, the request is being forwarded to only one of its neighbouring computers. The process continues until the data is found or the terminating conditions are met. The terminating conditions are met when the request has been forwarded for a preset number (TTL) of times or when the request originator has got a response from another node. As the request is being forward for a certain number of times (another preset number), the computer receiving the request contacts the request originator to see if it has received a response from another node. Therefore, instead of trying to find all copies of the same data in the P2P network, k-Walker Random Walk aims to find a single copy of the data needed. Another important difference between the k-Walker Random Walk and the BFS variants is that duplicate requests will not be ignored. Lv, et al.'s [3] experiment shows that the k-Walker Random Walk poses a 99% decrease in bandwidth requirement compared to BFS while posing only a slight increase (2% to 11%) in response time to that of BFS in all random network topologies.

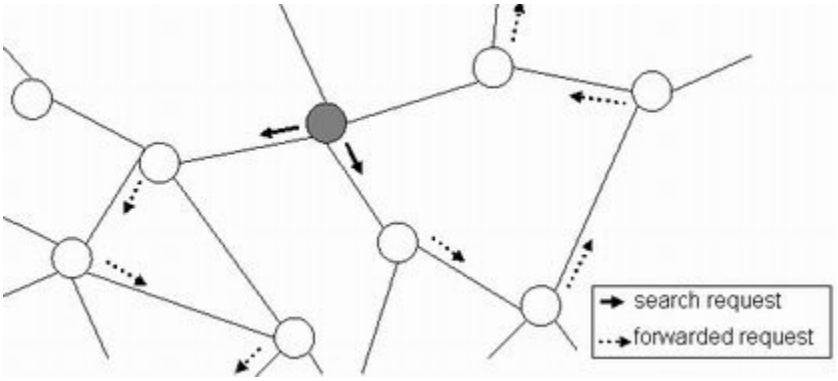

Figure 3. Randomized BFS with k set to 2.

Due to the varying conditions the experiments are conducted for the different search techniques, the experimental results cannot be compared. This paper tries to overcome this variation by simulating these search techniques in the same setting.

IV IMPLEMENTATION DETAILS

A. Simulation Core

1. Implementation of the Search Strategies in NS

I have implemented the application logics for the different search strategies as individual C++ classes. Only a few functionalities that are not known until run time are exposed to the OTcl side. By decreasing the amount of OTcl calls and leaving the core of the implementation in C++, the performance of the simulations will be maximized. Performance is an important issue because a maximum of 500 nodes with around 7500 links performing 1000 searches will be simulated. The following diagram shows the class hierarchy for the different search classes.

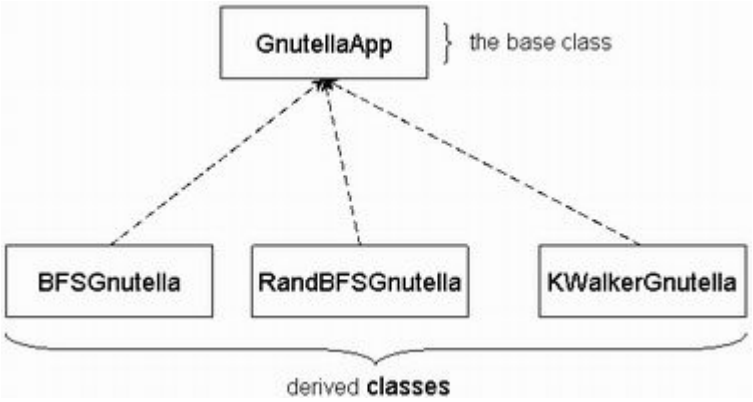

Figure 4. Class hierarchy for the search classes.

The BFSGnutella, RandBFSGnutella and KWalkerGnutella implements BFS, Randomized BFS and k-Walker Random Walk respectively. All of them need to implement two virtual methods inherited from GnutellaApp: *search()* and *process_data()*. The *search()* method instructs the Gnutella application to initiate a new search. It creates a unique id for the search request, initiates the TTL and other parameters for the search request and sends the request to its neighbours. The neighbours selected are depended on the class of the GnutellaApp (e.g. BFSGnutella selects all neighbours while KWalkerGnutella randomly selects k neighbours for a pre-specified k.). The *process_data()* method is a callback procedure when network messages (in this case search requests) are received. The method gets the search request, performs duplicate processing, checks if the file requested is present, decrement TTL and forwards the request to its neighbours. The three classes vary in the way they perform in the above processing. In the k-Walker Random Walk implementation, the periodic check to see if the originator has got a response from another node is not implemented. The feature can

be implemented easily but the omission is discovered too late that it is impossible to rerun all the simulation scenarios in time (takes about 2 days). Therefore, I decided to drop the feature for this project. For details, please refer to the source code.

The base class GnutellaApp is inherited from the NS class Process that provides functionalities for applications to receive and process network messages. GnutellaApp provides an interface for and implements the common functionalities shared by its derived classes. Files located in the current node are represented as integers stored in a data list. Each search request has a unique id that is stored in a received list to detect duplicate search requests. All lists used in my implementation belong to the class MyList which is a list class I implemented to customize for the specific needs for the simulation. The TTL and message size for a search request can also be specified in GnutellaApp. Utility methods like randomly choosing a preset number of neighbours for use in searches are also implemented.

2. TCP Connection Management and Message Passing

NS only allows a single TCP connection between two applications normally. However, in our simulation, each Gnutella application needs to maintain multiple connections to its neighbours. To overcome this limitation, a modified version of the approach in the webcache example provided by NS is adopted. The following diagram depicts the class structure for the connection management.

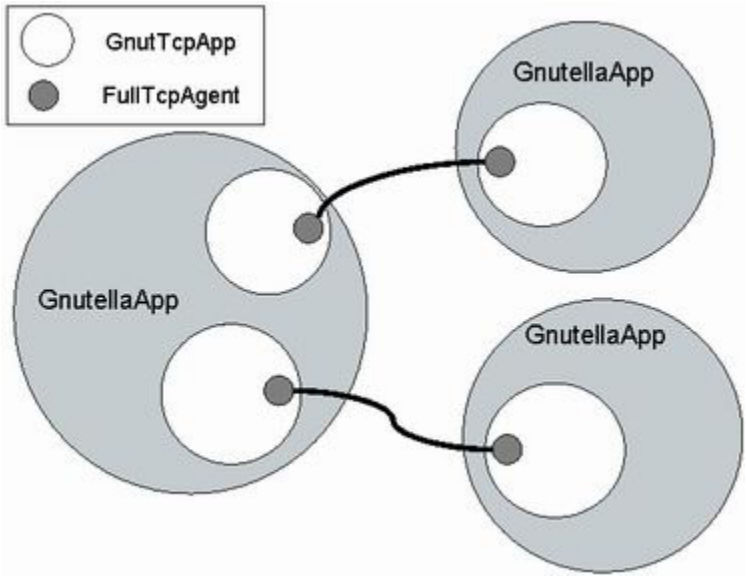

Figure 5. Class relationships for TCP connection management.

Each GnutellaApp has one or more GnutTcpApp objects stored in a list. Each of them is responsible for a single two-way TCP connection with one of the neighbours of the current GnutellaApp object. This overcomes the single connection limitation imposed by NS Application objects. GnutTcpApp acts as the middle man between GnutellaApp and TcpAgent.

Each search request is an instance of the struct gnut_data that contains all the control information for the search. In order for applications to exchange search requests, a wrapper class, GnutTcpAppData derived from AppData that fits the interface provided by Process (GnutellaApp's superclass) is needed. GnutTcpApp is responsible for embedding gnut_data inside a GnutTcpAppData and send the request out to another node. It is also responsible for keeping track of the amount of network traffic (in number of bytes) received from the neighbour node and passes the data to GnutellaApp when a complete message has been received. Therefore, GnutTcpApp is the class who performs the real job of sending and receiving search requests.

3. Data Collection

I have implemented the StatTrace class for collecting simulation data for statistical purposes. The StatTrace class is implemented as a singleton (a singleton is a class where only a single instance will be created). Any object can access it by a call to *StatTrace::instance()*. The class contains an instance of the struct stat_data that stores various information (e.g. the amount of traffic at the moment) for the simulation. StatTrace also contains an instance of the class StatTraceTimer derived from TimerHandler which writes out stat_data to a csv file in a comma delimited file format that can be read by Excel for analysis. A Tcl_Channel instance is used to perform the file write.

4. Data Structures

There are a number of data structures implemented to facilitate our simulations.

stat_data

It records the following information:

search_count – total number of search requests originated

srch_msg_count – amount of network traffic (in number of messages) in the network at an instance

total_srch_msg_count – accumulated amount of network traffic (in number search messages) since the beginning

success_count – total number of files found by the searches

dupl_count – total number of duplicated search requests received by all nodes

gnut_data

It contains the following information:

id – unique id for this search request

file – the file this search is aiming for (as an integer)

ttl – TTL for this search request (decremented each hop)

from – the GnutTcpApp that this search request is received from

origin – the GnutellaApp that originates this search

Basically, files are integers in the range from 1 to the number of nodes to reduce processing time, simplify implementation and simplify file generation that will be discussed in section C.

B. Visualizing the Simulations

Nam [8] is used to visualize the simulation. It is useful for small node counts. For large node counts (> 100 nodes), the nodes and links are so packed that it becomes useless. Therefore, nam files, which are read by Nam, are only generated for simulations with small node counts. For the ease of viewing, packets are color coded as red while nodes are colored as green when a node originates a search, blue when a node has the file a search request is looking for, and black whenever a search request's ttl becomes 0 or whenever it does not have a file a search request is looking for.

In visualizing the Gnutella simulations, three problems are encountered: (i) TCP handshake packets obscure the visualization of search requests propagations; (ii) packets become too small to see in links with large delays; (iii) changing colors for the packets and nodes requires a large number of OTcl calls that affects performance and stableness of the simulations.

To solve the first problem, I have modified main.cc and packet.cc under nam-1.9 to allow Nam to accept an argument "-w" that suppresses it to animate packets that are smaller than or equal to 40 bytes in size. Those packets are TCP handshake packets containing no data other than the TCP and IP headers. For the second problem, I modified packet.cc again to set the minimum length of a packet to be equal to its width. The third problem requires code modifications in both Nam and NS. Under Nam, I have hardcoded the packet's color to red. This eliminates the need for having an OTcl call for each link in order to change the packets' colors to red. To change the color of a node, a node trace is needed to be written to a nam file. However, a node cannot write to the nam file unless the OTcl command *namattach* is called for that node object to instantiate its *Tcl_Channel*. To eliminate a *namattach* call for each node object, I have modified node.cc under NS's common directory to change its *Tcl_Channel* to static variable. In this way, only a single *namattach* call is needed.

C. Topologies, Files and Search Requests Generations

Topologies for the simulations are generated by Georgia Tech Internetwork Topology Models (GT-ITM) generator [6] that comes with the NS-2.26 all-in-one package. GT-ITM generator reads from a parameter file for the generation algorithm and parameters that affect the average out-degree of the nodes and other characteristics of the network to generate the topology. In my simulations, Waxman 1 is the selected algorithm being used. I have chosen Waxman because according to Zegura, et al. [7], Waxman is "perhaps the most common generation method". The topology generated is in sgb file format. The utility program, sgb2ns, provided by the NS package will read the sgb file and convert it to OTcl script format. In our simulation, each node will have a GnutellaApp associated with it and each duplex link will have two FullTcpAgents that are connected and in listen state and two GnutTcpApps associate with them. Also, the GnutTcpApp on each side has to be added to a GnutellaApp associated with the node on that side of the connection. Therefore, I have modified the source file sgb2ns.c to add the above operations automatically and in OTcl script format. Another useful utility provided by GT-ITM is the edriver program. It reads the sgb file generated and reports the average

out-degree for the nodes in the network. I have been using this program to tweak the parameters in the parameter file to suit my simulation needs.

Besides the topology, the files located in each node in the simulation needs to be generated automatically. Therefore, I have written a C++ utility program named filegen that accepts the maximum number of files (maxFiles) a GnutellaApp can contain as an argument and randomly generates the files for each GnutellaApp object. For each GnutellaApp object, filegen will first generate the number of files this object will contain (between 1 and maxFiles). Then it generates the random files for that GnutellaApp object and makes sure that the same file will not appear twice in a single application object. The output file is in OTcl script format.

I have also written a C++ utility program, searchgen, to generate search requests for the simulation. Given the number of nodes, the number of search requests needed and the duration of the simulation, search requests (as triplets of time, originator and file to look for) will be randomly generated. searchgen will make sure the same node will not make the same search request twice in a simulation unless it is unavoidable. The unavoidable condition happens when the number of search requests exceeds the square of the number of nodes. In this case, it is impossible to not have duplicate search request as the number of unique pairs, originator and file to look for, run out. The output file is in OTcl script format.

D. Simulation Scenarios and Helper bash Scripts

Simulations for low and medium networks are run for all combinations of the number of nodes (in 50 and 100), the number of searches (in 50, 100 and 500) and the durations (in 10 and 100 seconds) with a TTL of 4. For high node counts, simulations for 500 nodes performing 500 and 1000 searches in 10 and 50 second respectively with a TTL of 7 are executed. Each simulation scenario are run for 5 times both on dense network (with high average out-degree) and on sparse network (with low average out-degree). The output files are named in the form “<search method><number of nodes>-<number of searches>-<duration><d for dense and s for sparse><trial count>.csv” without the “<” and “>”. For Randomized BFS, 50% is used as the percentage of neighbours a search request will be forwarded to. The percentage is the same as the one

chosen by Kalogeraki, et al. [2]. For k-Walker Random Walk, I arbitrarily pick 3 for k. I have also chosen the search request packet size (including the TCP/IP headers) to be 85 bytes that can be broken down as follows (slightly modified from the example in [4]):

IP header	**20 bytes**
TCP header	**20 bytes**
Gnutella header	**23 bytes**
Minimum Speed	**1 byte**
Search string	**20 bytes + 1 byte** *(trailing null)*
Total:	**85 bytes**

Table I. Search request packet makeup.

To ensure fairness for the simulations, the network topologies, the files contained in each GnutellaApp and the search requests will be generated once and used across all search strategies and trials.

I have written an OTcl script, gnut-sim.tcl that can take arguments to perform simulations on different scenarios. The usage of it is as follows:

```
Usage: ns gnut-sim.tcl [-method search_method] [-grid] [-node count] [-
dense] [-sparse] [-searches count] [-duration t] [-ttl ttl] [-fraction f]
[-k k] [-req-size s] [-trial tr]
```

Please refer to the source code for details.

I have also written a number of bash scripts to help in automating the simulation process. The genall script calls the topogen, filegen and searchgen scripts to generate all the topologies, files locations and search requests for our simulations needs. The batch-sim script calls the batch-sim-low and batch-sim-high to perform the actual simulations for low node counts and high node counts. To start the simulations, first run genall followed by batch-sim.

E. Statistics Helpers

I have implemented the C++ program combinecsv to combine the csv files generated by the same simulation scenario for the 5 trials. The bash script, batch-combine, will automate the process for all scenarios. After the combination, the csv files will be opened in Microsoft Excel to calculate the average values for the five trials.

I have written another C++ program furthercombine to further combine the csv files that have been processed by Excel for the three different strategies. After that, the further combined files can be opened in Excel again to generate graphs and statistics data for analysis and interpretation.

V RESULTS AND DISCUSSION

After studying the simulation results, I have discovered that the values of k and ttl used for the k-Walker Random Walk are too low for all the simulation scenarios. The results, which will be listed later in this section, suggest that while the k-Walker Random Walk generates very small amount of network traffic, its success rate is way too low for comparison with other search techniques. Therefore, extra simulation scenarios are performed to test the appropriate values of k and ttl used for the simulations. Since the scripts I have written to do the extra tests are used in disposable manner, they will not be included in the code listings.

Because of the large number of simulations results generated, I have selected the following ones for discussion purposes:

Simulation Scenario	**Nodes Count**	**Average out-degree**	**Number of searches**	**Duration**	**TTL**
A	50	2.44	100	10	4
B	50	3.24	100	10	4
C	100	4.46	500	100	4
D	100	4.48	500	100	4
E	500	5.036	1000	50	7
F	500	14.92	1000	50	7

Table II. Simulation scenarios used for discussion.

Scenarios A and B are categorized as small networks; C and D are categorized as medium networks; and E and F are categorized as large networks.

The results for the small network simulations are listed as follows:

	BFS	**Rand BFS**	**k-Walker k(3)**	**k-Walker k(4) ttl(6)**
Success Rate	0.91	0.46	0.266	0.484
Messages/Search	30.54	13.946	11.5	22.62
Duplications/Search	6.01	1.224	5.756	13.912
Max messages at an instance	38	11.8	11	20.8

Table III. Simulation results for scenario A.

	BFS	**Rand BFS**	**k-Walker k(3)**	**k-Walker k(4) ttl(6)**
Success Rate	1.26	0.734	0.322	0.624
Messages/Search	55.45	27.064	11.436	22.268
Duplications/Search	19.06	5.384	4.466	10.934
Max messages at an instance	90	31.8	13.8	23.2

Table IV. Simulation results for scenario B.

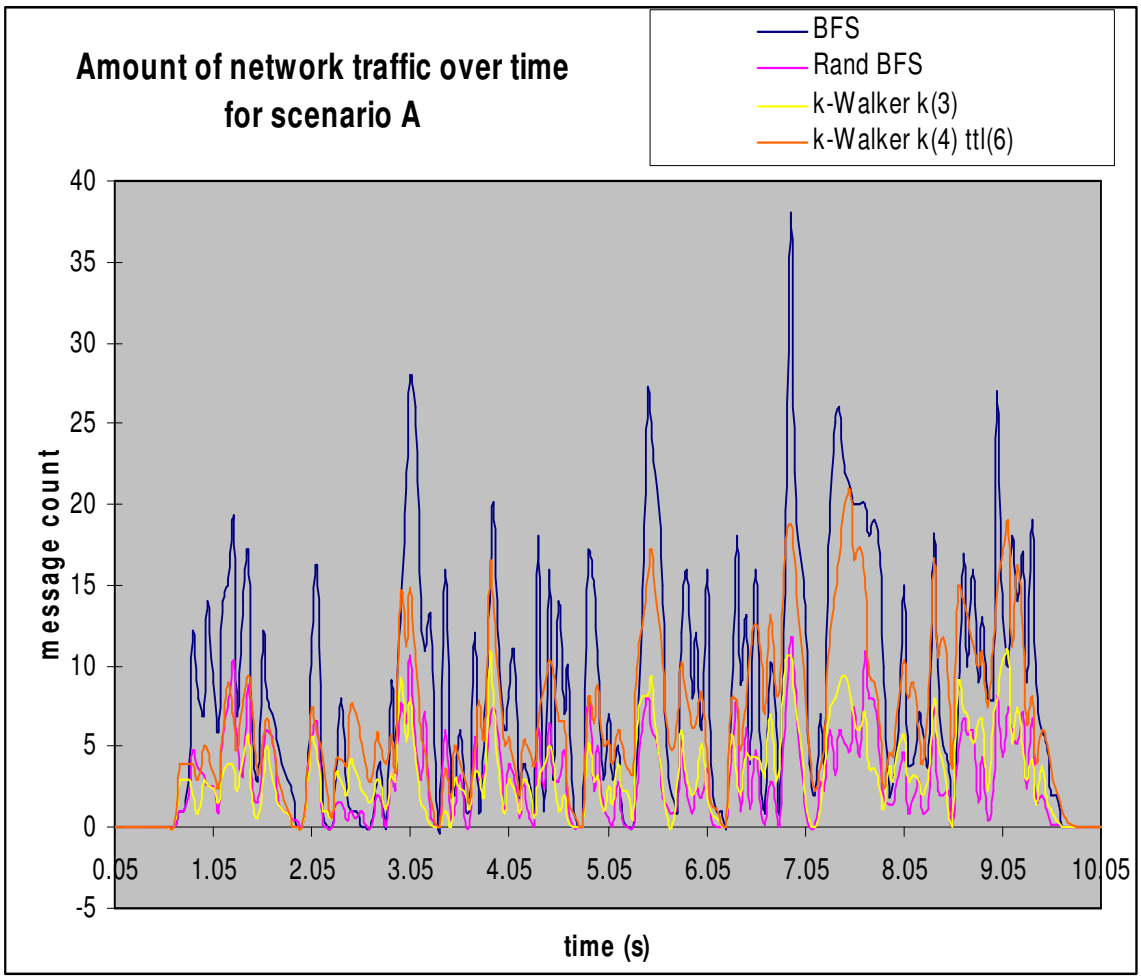

Figure 6. Amount of network traffic over time for scenario A.

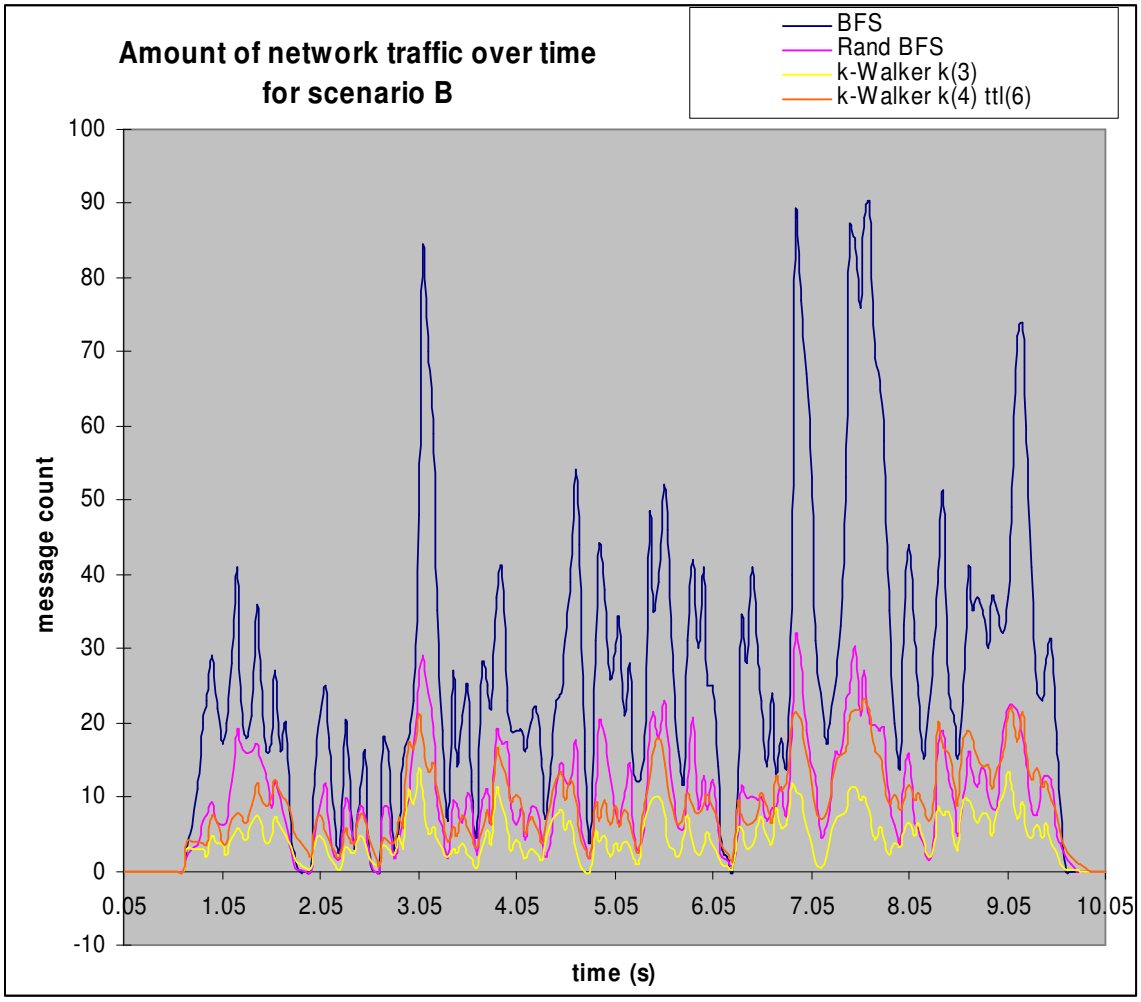

Figure 7. Amount of network traffic over time for scenario B.

For the small network scenarios, we can see that while the Randomized BFS and k-Walker Random Walk generates less amount of network traffic, their success rates for searches also decrease. The amount of traffic generated seems to be proportional to the success rate in these scenarios. Therefore, we do not see any advantages of using one search strategy to another in terms of success rate with respect to the amount of traffic generated. However, the k-Walker Random Walk with a higher k and TTL values results in more than double the amount of duplicate messages than the BFS in sparse network. In a small and sparse network, the k-Walkers have no choice but to repeat the steps and back track often when it reaches the network edge before the TTL expires. Therefore, we can conclude that a k-Walker Random Walk with a high k and TTL is not preferred in a small and sparse network.

The following are the results for the medium network scenarios:

	BFS	Rand BFS	k-Walker k(3)	k-Walker k(6) ttl(8)
Success Rate	1.486	0.858	0.228	0.7076
Messages/Search	147.122	59.5724	11.6032	45.1228
Duplications/Search	67.616	13.4368	2.9688	19.1028
Max messages at an instance	271	88.2	15.4	47.6

Table V. Simulation results for scenario C.

	BFS	Rand BFS	k-Walker k(3)	k-Walker k(6) ttl(8)
Success Rate	1.488	0.8624	0.1928	0.6828
Messages/Search	153.924	60.784	11.672	45.3772
Duplications/Search	71.876	13.1204	2.8588	18.3892
Max messages at an instance	332	90.8	15.6	42

Table VI. Simulation results for scenario D.

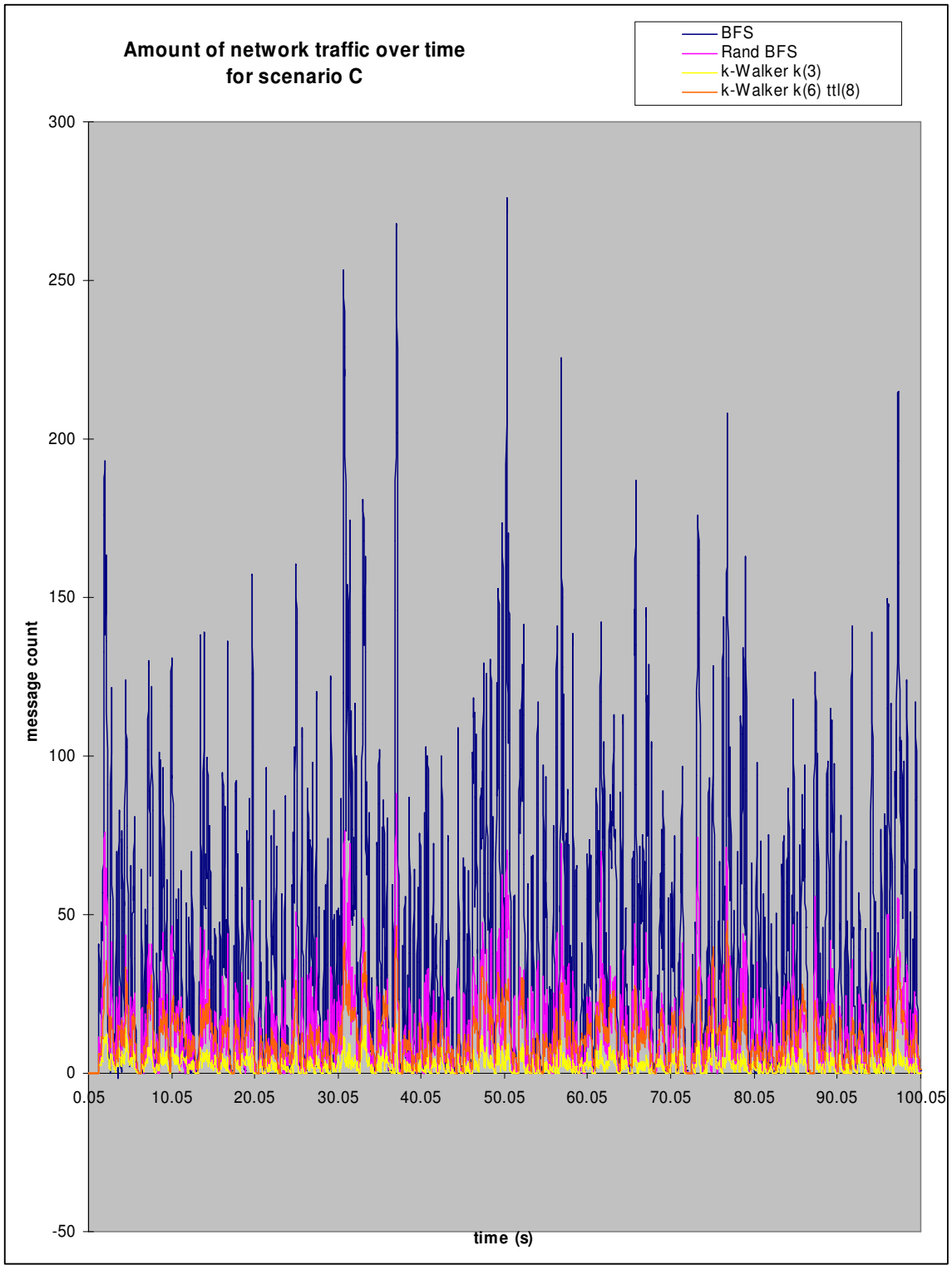

Figure 8. Amount of network traffic over time for scenario C.

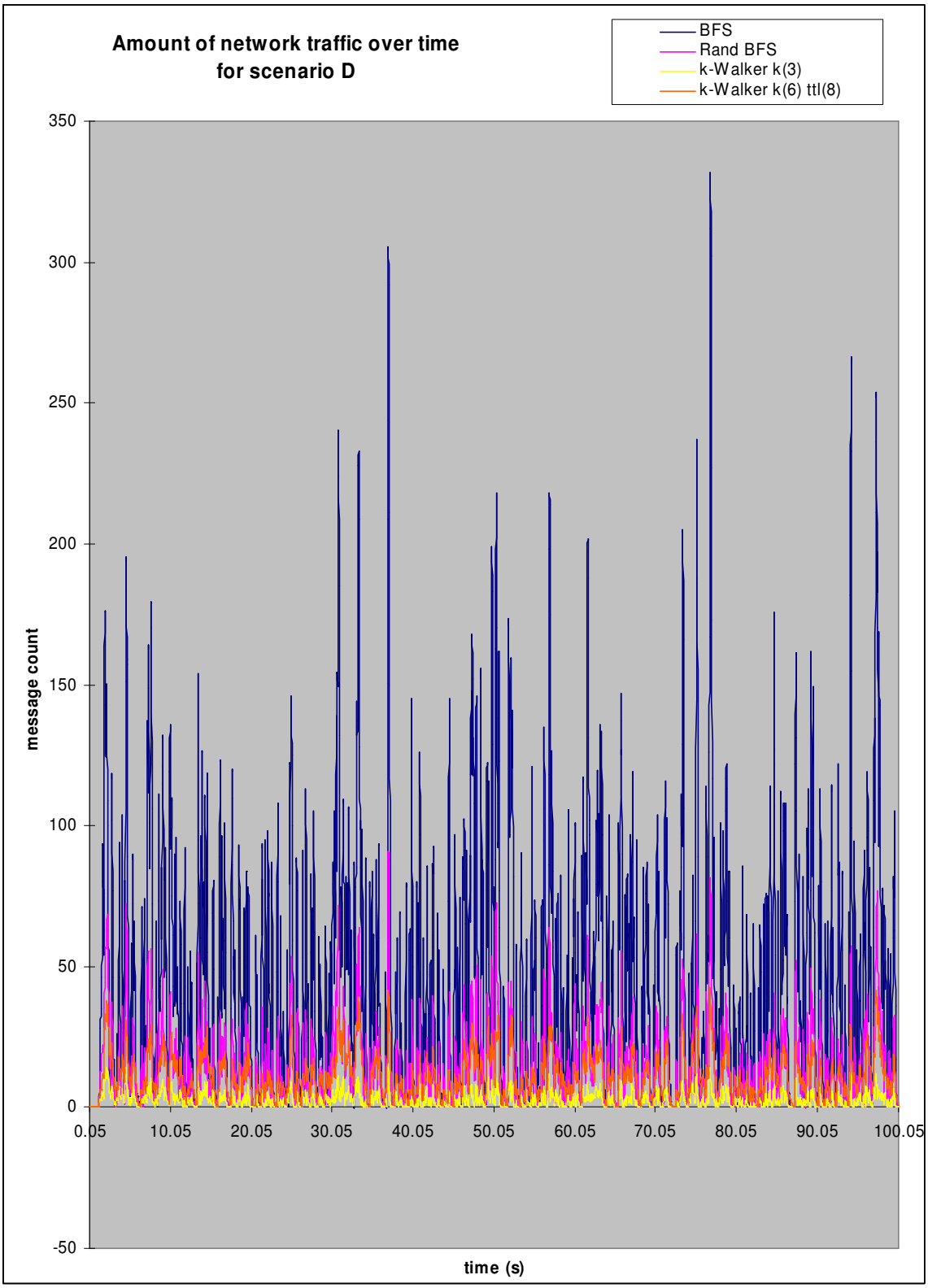

Figure 9. Amount of network traffic over time for scenario D.

The medium network scenarios are where the alternative search strategies show their advantages over BFS. In BFS, the success to the amount of traffic per search ratio is around 0.01 in both dense and sparse network. However, the same ratio for Randomized BFS and k-Walker Random Walk are around 0.014 and 0.015 respectively in both dense and sparse network. We can see that the Randomized BFS and k-Walker Random Walk pose a 40% and 50% improvements compared to BFS in terms of success rate to the amount of traffic ratio. Similarly, the improvement of Randomized BFS to BFS is even more in terms of success rate over maximum amount of traffic at an instance. However, k-Walker Random Walk is the king here since its maximum amount of traffic at an instance is only half of that of Randomized BFS. Therefore, we can say that the k-Walker Random Walk is slightly better than Randomized BFS and that BFS is the worst strategy in medium sized networks.

For large networks, the results are as follows:

	BFS	**Rand BFS**	**k-Walker k(3)**	**k-Walker k(15) ttl(13)**
Success Rate	3	2.3488	0.1048	0.929
Messages/Search	1431.759	669.1938	20.6566	189.12
Duplications/Search	945.618	284.0446	4.841	79.1965
Max messages at an instance	9344	2967	73.4	511

Table VII. Simulation results for scenario E.

	BFS	**Rand BFS**	**k-Walker k(3)**	**k-Walker k(15) ttl(13)**
Success Rate	3.081	3.073	0.129	1.0825
Messages/Search	5433.324	2777.311	20.572	188.384
Duplications/Search	4934.326	2279.636	1.485	42.7065
Max messages at an instance	41844	16380	97	536.5

Table VIII. Simulation results for scenario F.

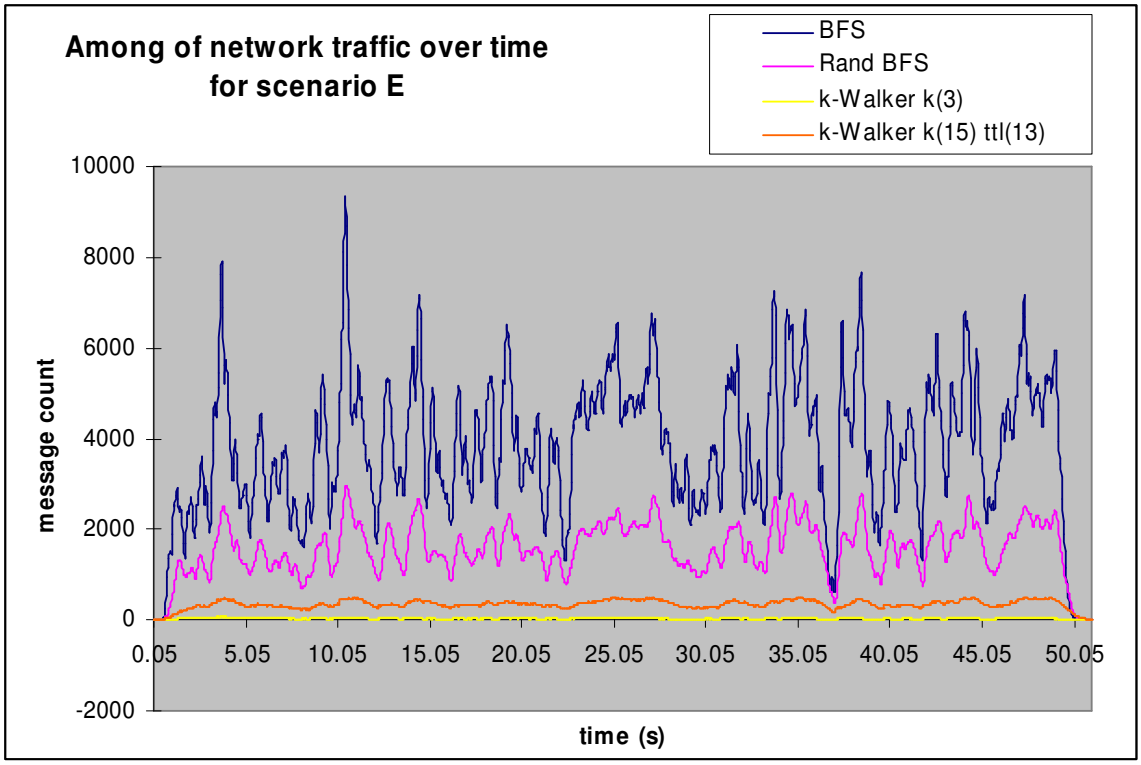

Figure 10. Amount of network traffic over time for scenario E.

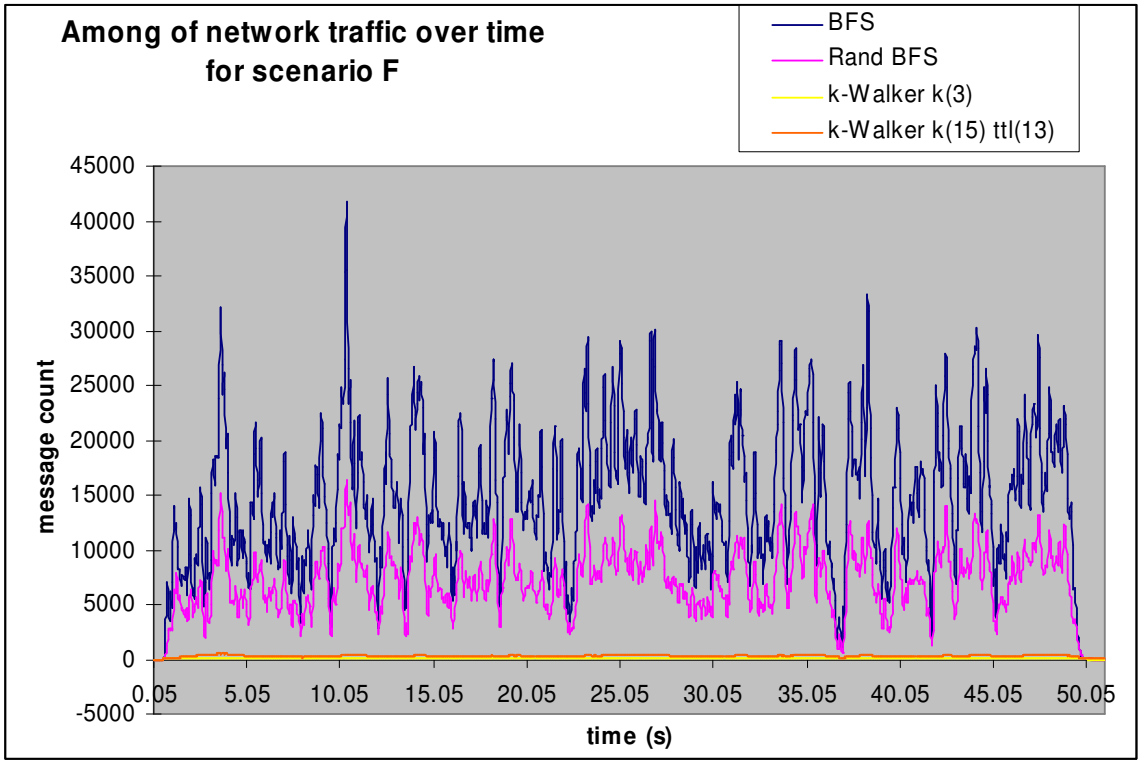

Figure 11. Amount of network traffic over time for scenario F.

The large network scenarios are where the k-Walker Random Walk obviously works better. From the above network traffic over time graphs, we can easily see that while the Randomized BFS generates about half of the traffic with to that of BFS, the k-Walker Random Walk only generates around 10% and 3% to that of BFS in sparse an dense network configuration respectively. Looking at the success rate of the searches, Randomized BFS has a success rate that is 78% to that of BFS in sparse network and has almost the same success rate as the BFS in dense network generating only half the amount of network traffic as the BFS. Although k-Walker Random Walk has a lower success rate when compared to the other search techniques, it still has the best success rate to amount of network traffic ratio. Not only that, it has achieved 0.93 and 1.08 success rate for the sparse and dense network configurations respectively. This means that the k-Walker Random Walk almost guarantees to find a single copy of a data needed and that is the goal of the k-Walker Random Walk. With a satisfactory success rate and the extremely little amount of traffic generated as compared to other strategies, k-Walker Random Walk is the best search technique while Randomized BFS sits in the middle and BFS, the worst, in large scaled networks.

VI CONCLUSIONS

In this project, three search strategies for Gnutella have been successfully implemented in NS. Nam has been used to visualize and verify the implementation. GT-ITM topology generator has been utilized to generate random topologies that more closely mimic a real internetwork. I have also used OTcl scripts and bash scripts to automate the simulations for many different configurations.

Simulation results have shown that the performances of the three different search techniques do not vary much in small networks. However, the k-Walker Random Walk stands out as the best technique in large networks. The Randomized BFS sits in the middle ground. And the BFS is the worst in large networks. We can, therefore, conclude that the search techniques suggested by the literatures are valid and will improve the search experience as the network gets larger.

VII FUTURE WORK

The first attempt in future work would be to implement the periodic check with the originator node whether a search response has been received from another node in k-Walker Random Walk. This feature is missing in the current implementation. The next step could be to implement extra search strategies for comparison. Another possible research direction would be to implement Gnutella 0.6's Ultrapeer into the network topologies.

If time permits, it would be interesting to look at GnutellaSim by He, et al. [9]. GnutellaSim is an NS-2 packet-level Gnutella simulator built as part of the work of the COMPASS group in the College of Computing at Georgia Tech, the institute that creates GT-ITM.

VIII REFERENCES

[1] Protocol Specification: Clip2 (n.d.). The Gnutella Protocol Specification v0.4. Retrieved January 1, 2003, from http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[2] Series Proceeding Section Article: Kalogeraki, V., Gunopulos, D., & Zeinalipour-Yazti, D. (2002, November). A Local Search Mechanism for Peer-to-Peer Networks [Electronic version]. Series-Proceeding-Section-Article, 300-307.

[3] Conference Paper: Lv, Q., et al. (2002, June). Search and Replication in Unstructured Peer-to-Peer Networks. Paper presented at the 16th International Conference on Supercomputing. New York City, NY.

[4] Online Article: Ritter, J. (2001, February). Why Gnutella Can't Scale. No, Really. Retrieved January 1, 2003, from http://www.darkridge.com/~jpr5/doc/gnutella.html

[5] Web Page: Gnutella Protocol Development (n.d). Retrieved October 30, 2003, from http://rfc-gnutella.sourceforge.net/index.html

[6] Web Page: Modelling Topology of Large Internetworks (n.d.). Retrieved December 1, 2003, from http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html

[7] Journal Article: Zegura, E. W., Calvert, K., & Donahoo, M. J. (1997). A Quantitative Comparison of Graph-based Models for Internet Topology [Electronic version]. IEEE/ACM Transactions on Networking. Retrieved November 1, 2003, from http://citeseer.nj.nec.com/zegura97quantitative.html

[8] Web Page: Nam: Network Animator (n.d.). Retrieved November 1, 2003, from http://www.isi.edu/nsnam/nam/

[9] Web Page: He, Q., et al. (2003, September). Packet-level Peer-to-Peer Simulation Framework and GnutellaSim. Retrieved December 4, 2003, from http://www.cc.gatech.edu/computing/compass/gnutella/

APPENDIX 1: CODE LISTING

The following code listing contains the files:

- OTcl Files
  - under ~/gnutella
    - gnut-sim.tcl – the main entry points for the simulations
  - under ~/gnutella/autogen/grid
    - grid16.tcl – a topology of a 4x4 grid used for verification of the Gnutella simulation
- C++ Files
  - under ns-allinone2.26/ns-2.26/gnutella
    - gnutellaapp.h – the interface for all GnutellaApps
    - gnutellaapp.cc – implements the common functionalities of all GnutellaApps
    - gnutella-bfs.cc – implements BFS GnutellaApp
    - gnutella-randbfs.cc – implements Randomized BFS GnutellaApp
    - gnutella-kwalker.cc – implements k-Walker Random Walk GnutellaApp
    - gnuttcpapp.h – header for GnutTcpApp
    - gnuttcpapp.cc – implementation for GnutTcpApp
    - mylist.h – header for MyList data structure
    - mylist.cc – implementation of MyList data structure
    - stattrace.h – header for the data collecting class
    - stattrace.cc – implementation of data collecting class
  - under ns-allinone-2.26/ns-2.26/common
    - node.cc – slightly modified to reduce the number of OTcl calls
  - under ns-allinone-2.26/nam-1.9
    - packet.cc – slightly modified to change the default packet color, change the size of packets and suppress handshake packets animation

        - packet.h – header slightly modified for color change
        - main.cc – modified to allow suppressing handshake packets animation
    - under ns-allinone-2.26/gt-itm/sgb2ns
        - sgb2ns – heavily modified to including various initializations for Gnutella simulations
    - under ~/gnutella/autogen/filegen/src
        - filegen.cc – generates files located in the nodes automatically
    - under ~/gnutella/autogen/search/src
        - searchgen.cc – generates search requests automatically
    - under ~/gnutella/combinecsv/src
        - combinecsv.cc – combines the results for different trials
        - furthercombine.cc – combines the results for different search techniques
- bash Scripts
    - under ~/gnutella
        - batch-sim – do batch simulations
        - batch-sim-low – do batch simulations for small and medium networks
        - batch-sim-high – do batch simulations for large networks
    - under ~/gnutella/autogen
        - genall – generates topologies, files locations and search requests used by all simulation runs
        - topo/topgen – generates topologies and produce statistics for the topologies used by all simulation runs
        - filegen/filegen – generates file locations for all simulation runs
        - search/searchgen – generates search requests for all simulation runs
    - under ~/gnutella/combinecsv
        - batch-combine – combines the results for different trials for all scenarios

- batch-further – combines the results for different search strategies for all scenarios