A Time Oriented Resource Locator for Unstructured Overlay Network

Wen-Fu Kao, Ying Yu Tai and Howard Che-Hao Chang

{wkao, yytai, hcchang}@ece.ucdavis.edu

Department of Electrical and Computer Engineering

University of California, Davis

March 2003

Abstract

Decentralized and unstructured peer-to-peer networks over existing internet infrastructure are becoming more and more popular because no central directory server is required. However, the search of the requested resources under software such as Gnutella causes heavy traffic load due to the flooding nature among the internet. The performance can be improved if the applications can have more awareness of the topology and the network status of a large overlay network. Many works try to improve the search efficiency and minimize the traffic load. In this paper, we propose a new algorithm based on the response time, called Time Oriented Resource Locator (TORL), for searching the requested resource in the unstructured P2P networks. We simulated our algorithm in two different network topologies and compared with k walker random walk algorithm. The results demonstrate an efficient search and the potential for further improving the traffic load.

1. Introduction

Traditional information exchange relies on the awareness between fixed server and stationary computers through internet. With the amount of internet users grows rapidly, the exchange between arbitrary two peers is highly desired, thus requiring ubiquitous connectivity. A single site provides the information cannot satisfy users' need. Unstructured overlay networks have become a good way to share information without prior knowledge about the location of one's target. However, locating the desired information among the decentralized network is difficult. Applications for file sharing in peer-to-peer networks such as Gnutella [1] and Freenet [2] are attractive because they provide an easy-to-use platform to meet this need. The problem of these applications is they may increase the network traffic dramatically. For example, Gnutella protocol provides a decentralized structure for arbitrary two computers. It uses a simple search among joined computers within certain TTL. First, it checks if a computer has a specific resource requested by a user. Second, the computer sends the user's request to all its know computers, which it connects to. This simple strategy can easily locate the resource and share with each other. However, measurement data has shown P2P networks have caused significant impact on network traffic [3]. Therefore, many algorithms have been presented to improve the performance of the search over P2P networks. Our goal is to provide a simple and scalable algorithm, which utilizes limited topology information to provide more sensitivity to network status, improving the performance and the loading. The remainder of the paper is organized as follows:

Section 2 summarizes current related work. Section 3 introduces and describes the concepts behind TORL. Section 4 depicts, through simulation, our experimental model. The preliminary results and conclusions are drawn respectively in sections 5 and 6.

2. Related Work

There have been many researches on overlay networks. Several papers use distance or response time as the location information to construct a better route. Method proposed by P. Francis et al. is based on the estimation of the distance [4].Z. Fei et al. provided a server selection techniques, which minimized the response time. The clients measure the response time periodically [5]. Another similar topologically-aware approach is proposed by S. Ratnasamy et al., which is a distributed binning scheme based on the RTT information [6]. Although their binning algorithms produced a good performance, prior landmark information has to be provided. On the other hand, Lv et al. [7] proposed a different scheme called Multiple Random Walks algorithm, which provides good performance on both quickness of search and traffic load. The multiple random walks algorithm doesn't require any knowledge about the physical topology except the link between two nodes. Every walker in this algorithm will choose one of the linked nodes to query and repeat the procedure until the walker finds the resource or the pre-determined TTL reaches. This algorithm has almost the same performance as Gnutella's flooding scheme without introducing additional network traffic. We will compare our algorithm with the multiple random walks in latter section and discuss it.

3. Approach

Since constructing a complete topology for the decentralized network is not feasible, we want to choose a representative metric to approximate the actual topology. Many works tried to use hop counts to estimate the distance to improve the performance. However, this does not take the network loading and host lifetime into account. We choose round-trip-time (RTT) between two nodes as our metric instead of hop counts. The RTT information can represent both the distance and loading. Longer RTT means a longer routing or a longer delay. An infinite RTT usually represents the host is not alive in this network. Using the RTT, we are able to construct the virtual topology more precisely to an instance of the network status. Intuitively, the faster a node can respond to a request, the more likely we can shorten the search process.

Initially, every node collects up to k fastest response nodes as its virtual neighbors. We heuristically collect at most three nodes for virtual neighbor set in this experiment. The virtual neighbor set will change whenever a node collects the neighbor information again. Therefore, the virtual neighbor set actively reflects current network traffic and host loading, and lifetime of hosts. We set the expiration period of a virtual neighbor set to half hour or one hour.

After obtaining the virtual neighbor sets, a starting node, which sends a request to the overlay network at certain time, begins to build a search tree until it finds the requested resource or the pre-determined TTL reaches zero. The search process is distributed and breadth-first generally. For

every parent node in level *n*, it queries its three neighbors in level *n+1*. If any one of the neighbors has the requested resource, this search path stops. Otherwise, the three nodes in level *n+1* send the query to their own neighbor nodes in level *n+2* simultaneously. Notice that when any one of the search reaches the goal, other search paths do not realize it. Therefore, the whole search will stop when all paths reach the pre-determined TTL in the worst case. A node only sends the query to its neighbor nodes except its parent node. We don't record all the path information along any individual search path, thus having the possibility to query a node's parent's parent node.

This *k*-nary search tree grows significantly when *k* is large, thus choosing the TTL value is very important.

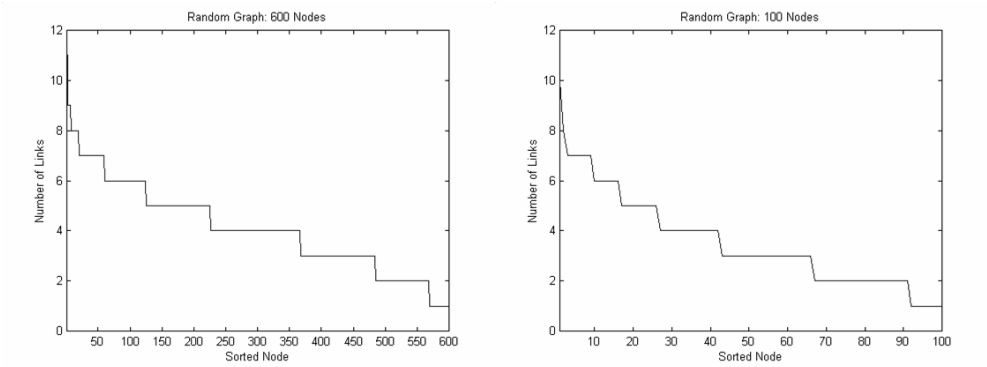

Figure 1

4. Experiment Model

We use GT-ITM [8] to generate graphs used in our simulation. The link cost generated by GT-ITM is used as the response time. The larger the cost, the longer response time it has. Figure 2 shows the topology we used in our experiment. We use random 100 nodes and transit-stub 600 nodes as our testing graph. For simplifying the model, when we construct the virtual neighbor set, we only collect virtual neighbors within TTL=1 area. This makes each of our virtual neighbors has a direct physical connection to every corresponding node.

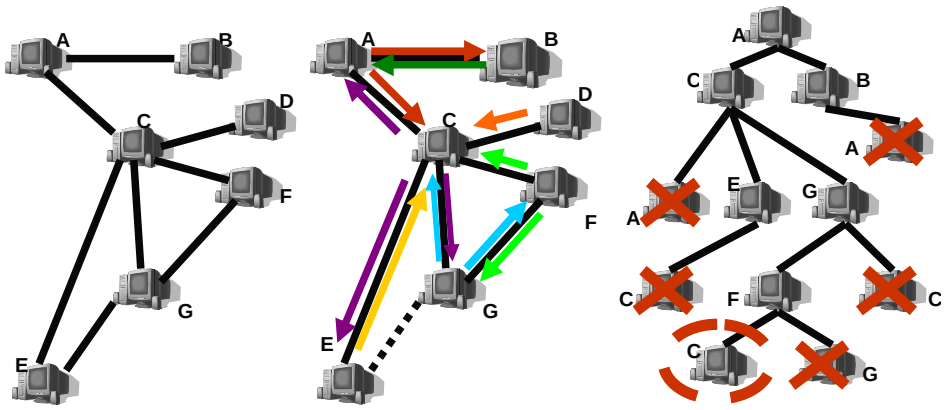

Figure 3

As show in Figure 4, the left graph shows the original topology. The middle graph is a directed graph after completing the virtual neighbor constructing. Node A only has two neighbor nodes and node C has more than three neighbor nodes. Since A, E, and G are the three fastest response node to C, it will only query them instead of D and F. As for node E, it only has one virtual neighbor since the connection between node E and node G is too slow. The right graph is a tree derived from the directed graph. Hosts with red-cross mark do not appear in the virtual neighbor set due to the host is both the parent and the child of one node. Notice that C (with big circle) is allowed to appear in F's virtual neighbor set. Although C already exists along the search path from starting node A to node F, F only recognizes its parent node G and doesn't have information about the entire search path, thus adding node C into its virtual neighbor set.

In Lv's experiment [7], random walk has the best performance in comparison with other searching strategies. We decide to compare our algorithm with k-walker random walk. Random walk is well-known searching technique, which chooses a neighbor to query randomly. Since random walk process chooses only one neighbor node, Lv increases the number of "walkers". A starting node sends k walkers and each walker is independently proceed its random walk. Lv's result shows 16 to 64 walkers usually achieve a good result. In our experiment, we use 12-walkers random walk.

We implemented our algorithm in C and 12-walkers random walk using MATLAB. Both simulations take the exactly the same graph. The requesting node and target nodes are generated randomly. Since we simulate one requesting process at one time, the requesting node is always one. Total target nodes are set to 5% of the number of the nodes in a topology. For ts600 topology, the total target nodes are 30. We use normal distribution with mean 5% to generate target nodes for 12-walkers random walk. As for our algorithm simulation, we use uniform distribution to generate 5% of target nodes. The random generator we used is from Paul Bourke, which uses a combination of a Fibonacci sequence and an "arithmetic sequence". The random number generator is based on a FORTRAN version from George Marsaglia and Arif Zaman, Florida State University.

5. Preliminary Simulation Results

We simulated both algorithms from TTL=1 to TTL=15. For every TTL, we run 200 simulations and calculated its average. Averagely, we found both algorithms can achieve a 50% hit rate within 2 to 4 TTL. In Figure 5, we compare the worst case of our algorithm and found out it still achieve more than 50% hit rate within 15 TTL. Based on the result of random-100 and ts-600 simulations, our algorithm becomes stable when the enough TTL meets. However, 12-walker random walk has a larger deviation even in a large TTL.

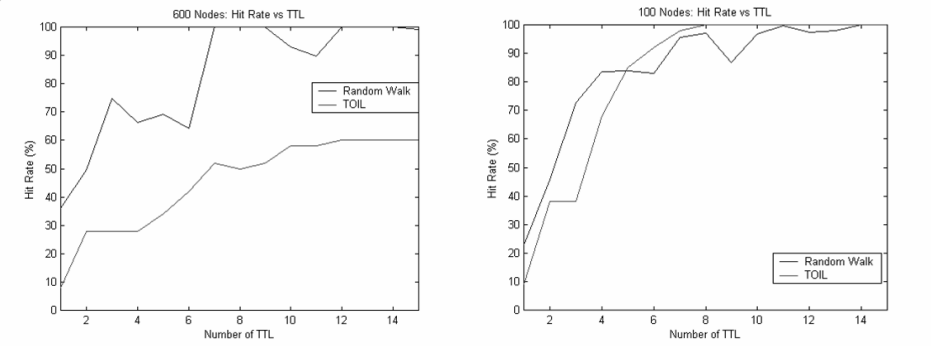

Figure 6

As for the traffic loading, as we expect, our algorithm grows in an exponential rate since we build a ternary search tree. Random walk algorithm has a better performance due to the linear growth. We take their logarithm vs. TTL in Figure 7. Both simulations show that the two algorithms have the same traffic loading when TTL=5. Although the tree grows exponentially, the loading is smaller than 12-walker random walk when TTL is less than 5. The reason is 12-walker random walk sends 12 query from the start while our algorithm only sends three and won't exceed 12 within two TTL.

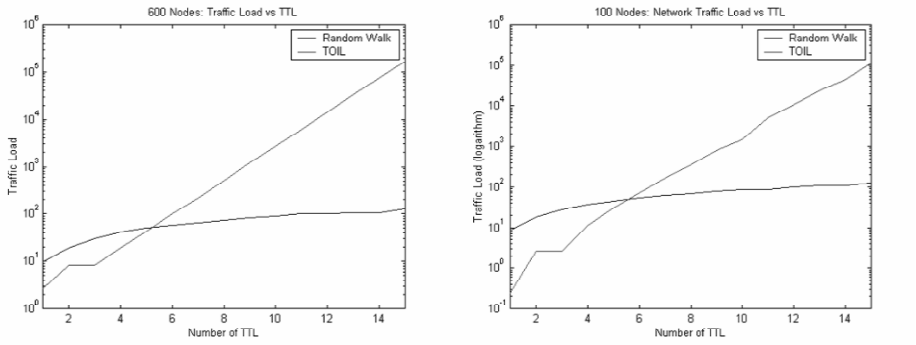

Figure 8

Since the reason we choose the response time as our metric for building the search tree is to minimize the search time of a certain resource, we want to know how much time we can save. Figure xx demonstrates the hit time ratio (HTR) in different TTL. We define the hit time ratio as the ratio of the two times:

$$HTR = \frac{T_t}{T_r}$$

$T_t = E$ [Hit Time | our algorithm finds the shortest path to the target for a given TTL]

$T_r = E$ [Hit Time | random walk finds the shortest path to the target for a given TTL]

From Figure 9, we see our algorithm only uses 40% of random walk search time to locate the requested resource except TTL=2. The reason our algorithm uses nearly 1.4 times of the random walk search time is the potential indirect routing. For example, if starting node has a virtual neighbor set B, C and D, and one of our target nodes is E. Node E is the first neighbor in B's virtual neighbor set. Since

we always choose the smallest response time first, the possible search path would be A-B-E. However, it is possible that there is actually a longer link between node A and node E. Due to the response time from A to E is slower than the other three neighbor nodes, node A wouldn't add E into the set. Consequently, the total link of A-B-E is greater than A-E, thus having a longer hit time. Random walk works well in this situation because it chooses a node randomly. This gives walker a greater possibility than our algorithm to select E rather than fastest response nodes. Although this situation could happen, we found its impact of the search time is limited due to the possibility of this case is low.

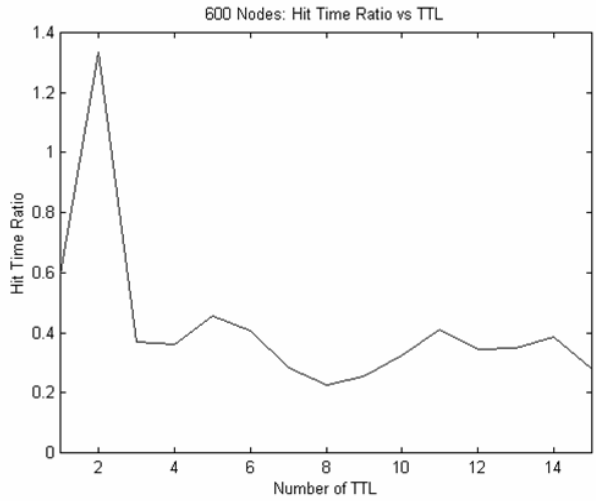

Figure 10

6. Conclusions and Future Work

This is our preliminary experiment. Our simulation shows the potential of this algorithm. We effectively decrease the search time using the RTT as the locality information. The only sacrifice is a short burst of the traffic within certain nodes. As long as a network can tolerate this short burst of the traffic, this algorithm works well. Our algorithm doesn't require complicated routing information and only requires a small effort to build each node's virtual neighbor set. The performance we obtained depends on the topology much. Since this algorithm tends to locally optimize the search time, it is very suitable for transit-stub topology. In real world, network topology is similar to the transit-stub topology. We still need to simulate this algorithm on more topologies such as PLRG to evaluate its performance change. We expect this algorithm should be more adaptable to different topology.

The number of virtual neighbor set is the main reason causing the traffic load. To minimize this traffic, there are several possible ways to modify our algorithm. One straightforward method is changing the ternary tree to a binary tree. We can only collect two neighbor nodes. If two is enough to achieve a decent performance, we don't have to use the ternary tree. For the same TTL, 10 for example, the number of children of binary tree is much less than a ternary tree. An alternative modification is only two of the nodes are selected based on the RTT and the last one is selected randomly. Intuitively, this gives the algorithm a chance to avoid potential indirect routing and stop the search quicker. The third modification would be querying only one neighbor node after reaching a certain pre-determined

TTL. In this way, we limit the total query loading within certain area and isolate the traffic from other network. All three modifications require further simulation to be able to find out the best combination of these parameters. Further improvement of this algorithm is expected.

7. References

[1] Gnutella, "http://gnutella.wego.com"

[2] Freenet, "http://freenet.sourceforge.net"

[3] Stefan Saroiu et al. "A Measurement Study of Peer-to-Peer File Sharing Systems", *Proceedings of the Multimedia Computing and Networking, 2002*

[4] P. Francis et al., "An Architecture for a Global Internet Host Distance Estimation Service", *Proceedings of IEEE Infocom 1999*

[5] Z. Fei et al., "A novel server selection techniques for improving the response time of a replicated services", *Proceedings of IEEE Infocom 1998*

[6] S. Ratnasamy et al., "Topologically-Aware Overlay Construction and Server Selection", *Proceedings of IEEE Infocom 2002*

[7] Qin Lv et al. "Search and Replication in Unstructured Peer-to-Peer Networks", *ICS02*

[8["http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html

Appendix: Simulation Source Code

1. Topology processing in PERL (topology.pl)

```
## ----------------------
## EEC 289Q Project
## ----------------------
## File: topology.pl
## Version: 1.0
## Created Date: March 8, 2003.
## Updated Date: March 8, 2003.
## Function: This Perl Script Transfers
## Topology Files to Format we need
## ---------------------------------
#! /usr/bin/perl -w
require 5.004;
use strict "vars";
use FileHandle;
use Cwd;
my($fname, $line, $index, $i, $j);
my(@check, @dtable, @cost);
$fname="r100.txt";
if($ARGV[0] eq "-h") {
          Help();
          exit(-1);
}
for($i=0; $i<100; $i++) {
     for($j=0; $j<100; $j++) {
          $dtable[$i][$j]=0;
          $cost[$i][$j]=0;
     }
}
open(F, "$fname") || ReportError("Could Not
Open '$fname' for Reading");
flock(F, 2) || ReportError("Could Not Lock
'$fname' (Try Again Later)");
while(defined($line=<F>)) {
     chop($line);
     @check=split(/\s/,$line);
print "W\t $check[0] \t $check[1] \n";
     $dtable[$check[0]][$check[1]]=1;
print "S\t $dtable[$check[0]][$check[1]]
\n";
     $dtable[$check[1]][$check[0]]=1;
     $cost[$check[0]][$check[1]]=$check[2
];
     $cost[$check[1]][$check[0]]=$check[2
];
}
close(F);
my($tindex) = 1;
my($fname1)="link100.txt";
my($fname2)="cost100.txt";
open(F, ">$fname1");
for($i = 0; $i < 100; $i++) {
     for($j = 0; $j < 100; $j++) {
          print F "$dtable[$i][$j]\t";
     }
     print F "\n";
}
close(F);
open(F, ">$fname2");
for($i = 0; $i < 100; $i++) {
     for($j = 0; $j < 100; $j++) {
          print F "$cost[$i][$j]\t";
     }
     print F "\n";
}
close(F);
sub ReportError
{
     print "$_[0]\n";
     print "Please Check it Again !!\n";
     exit(-1);

}
```

2. K Random Walk in MATLAB

```
% EEC289Q Class Project

close all;
clear all;
link=load('link100.txt');
cost=load('cost100.txt');
z=300;
for ttl=1:15
    for t=1:z
```

```
N=100;
age=N/20;
sd=age;
O=ceil(N*0.05);
d=sum(link);
sd=fliplr(sort(d));
n=linspace(1,N,N);
object=zeros(1,N);
j=1;
    for i=1:O
        j=ceil(random('unif',0,N));
            while object(j)==1

j=ceil(random('unif',0,N));
            end
        object(j)=1;
    end
    sp=ceil(random('unif',0,N));
     while object(sp)==1
        sp=ceil(random('unif',0,N));
    end
k=12;
if(k>d(sp))
    k=d(sp);
end
for j=1:k
    pc(j,1)=sp;
    cp(j)=0;
    hop(j)=0;
    lcost(j)=0;
    check(j)=0;
    flag(j)=0;
end
for j=1:k

r(j)=ceil(random('unif',0,d(sp)));
    I=find(link(sp,:));
    double=0;
    for b=1:(j-1)
        if cp(b)==I(r(j));
            double=double+1;
        end
    end
    while double>0

r(j)=ceil(random('unif',0,d(sp)));
        double=0;
        for b=1:(j-1)
            if cp(b)==I(r(j));
                double=double+1;
            end
        end
    end
    cp(j)=I(r(j));
    hop(j)=1;
    lcost(j)=cost(sp,cp(j));
    nn(j)=d(cp(j));
    if (object(cp(j))==1)
        check(j)=1;
        flag(j)=1;
    end
    pc(j,2)=cp(j);
end
for i=1:(ttl-1)
    for j=1:k

rw=ceil(random('unif',0,nn(j)));
        I=find(link(cp(j),:));
        np=I(rw);
        if(object(np) == 0 & check(j) ==
0)
            hop(j)=hop(j)+1;

lcost(j)=lcost(j)+cost(cp(j),np);
            cp(j)=np;
            pc(j,i+2)=cp(j);
       elseif(object(np) == 1 & check(j)
== 0)
            hop(j)=hop(j)+1;

lcost(j)=lcost(j)+cost(cp(j),np);
            check(j)=1;
            flag(j)=1;
            cp(j)=np;
            pc(j,i+2)=cp(j);
        else
            cp(j)=np;
            pc(j,i+2)=0;
        end
        nn(j)=d(cp(j));
    end
end
for j=1:k
   if flag(j)==0
      pc(j,ttl+1)=cp(j);
   else
      pc(j,ttl+1)=0;
   end
end
success(t)=max(flag);
ihop(t,ttl)=min(hop);
thop(t,ttl)=sum(hop);
[a b]=sort(lcost);
y=0;
for x=1:k
  if flag(b(x))==0
  else
    if y>0
    else
      icost(t,ttl)=a(x);
      y=y+1;
    end
  end
end
if y==0
  icost(t,ttl)=0;
end
tcost(t,ttl)=sum(lcost);
end
rate(ttl)=sum(success)/z*100;
end
figure(1);
plot(n,sd);
title(sprintf('Random Graph: %d Nodes',
N));
xlabel('Sorted Node');
ylabel('Number of Links');
xlim([1 N]);
ylim([0 12]);
figure(2);
plot(rate);
title('Hit Rate vs TTL');
xlabel('Number of TTL');
ylabel('Hit Rate (%)');
figure(3);
plot(mean(thop));
title('Traffic Load vs TTL');
xlabel('Number of TTL');
ylabel('Traffic Load');
figure(4);
plot(mean(icost));
title('Cost vs TTL');
xlabel('Number of TTL');
ylabel('Link Cost')
```

3. Topology Processing in C++

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>

using namespace std;

void  retrieve_filename(char  in_name[],
char out_name[]);
    // Retrieve input and output filenames
that are input by user.
void  open_files(char  in_name[],  char
out_name[], ifstream& fin, ofstream& fout);
    // open files that are going to be read
and be written.
void close_files(ifstream& fin, ofstream&
fout);
    // Close Files that are opened.

const int node_amount = 100;

int main()
{
     char in_name[10],
         out_name[10];
```

```
     int node1, node2, cost, trash;
     int graph[node_amount][6];
     for(int i=0; i<node_amount; i++)
      {
        for(int j=0; j<6; j++)
         {
           graph[i][j]= 255;
         }
      }

     ifstream fin;
     ofstream fout;

     retrieve_filename(in_name,
out_name);
     fout.open(out_name);
     if(fout.fail())
      {
        cout<<"Failure   to   open   output
file";
        exit(1);
      }

     for(int  node_num  =  0;  node_num  <
node_amount; node_num++)
     {
        open_files(in_name, out_name, fin,
fout);
        char test;
        do
        {

          fin >>node1 >>node2 >>cost;
          char symbol;
          do
           {
            fin.get(symbol);
           }
          while(symbol != '\n');
          test=fin.peek();

          if(node1 == node_num && cost <
graph[node_num][5])
           {
             if(cost < graph[node_num][3] &&
cost <= graph[node_num][1])
              {

graph[node_num][5]=graph[node_num][3];

graph[node_num][4]=graph[node_num][2];

graph[node_num][3]=graph[node_num][1];

graph[node_num][2]=graph[node_num][0];
               graph[node_num][1]=cost;
               graph[node_num][0]=node2;
              }
            else        if(cost        <=
graph[node_num][3]     &&     cost     >
graph[node_num][1])
              {

graph[node_num][5]=graph[node_num][3];

graph[node_num][4]=graph[node_num][2];
               graph[node_num][3]=cost;
               graph[node_num][2]=node2;
              }
             else
              {
                graph[node_num][5]=cost;
                graph[node_num][4]=node2;
              }
           }

          else  if(node2  ==  node_num  &&
cost<graph[node_num][5])
           {
             if(cost < graph[node_num][3] &&
cost <= graph[node_num][1])
              {

graph[node_num][5]=graph[node_num][3];

graph[node_num][4]=graph[node_num][2];

graph[node_num][3]=graph[node_num][1];

graph[node_num][2]=graph[node_num][0];
               graph[node_num][1]=cost;
               graph[node_num][0]=node1;
              }
             else        if(cost        <=
graph[node_num][3]     &&     cost     >
graph[node_num][1])
              {

graph[node_num][5]=graph[node_num][3];

graph[node_num][4]=graph[node_num][2];
               graph[node_num][3]=cost;
               graph[node_num][2]=node1;
              }
             else
              {
                graph[node_num][5]=cost;
                graph[node_num][4]=node1;
              }
            }
          }
           while(test != EOF);
           fin.close();
     }

     for(int i=0; i<node_amount; i++)
      {
       for(int j=0; j<6; j++)
        {
          if(graph[i][j]==255)
            graph[i][j]=0;
        }
      }

     for(int i=0; i<node_amount; i++)
       {
       fout<<graph[i][0]<<"
"<<graph[i][1]<<" "<<graph[i][2]<<" "
          <<graph[i][3]<<"
"<<graph[i][4]<<" "<<graph[i][5]<<"\n";
       }

     fout.close();
     system("PAUSE");
     return 0;
}

void  retrieve_filename(char  in_name[],
char out_name[])
{
    cout<<"Please  input  ramdom  graph
name\n"
       <<"----->";
    cin >>in_name;
    cout<<"Please input output filename\n"
       <<"----->";
    cin >>out_name;
 }

void  open_files(char  in_name[],  char
out_name[], ifstream& fin, ofstream& fout)
{
    fin.open(in_name);
    if(fin.fail())
      {
        cout<<"Failure to open input file";
        exit(1);
      }
}
```

4. Time-Oriented Search in C

```
/* EEC289Q Term Project
/* Simulation for time based search for
overlay networks */
/* Author: Wen-Fu Kao */
/* Date: Mar 8, 2003 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "randomlib.c"
```

```
#define NODE_NUM 600
#define TGT 30
#define MAXTTL 10
#define SN 200
#define MAXSTACK 2048

int g[NODE_NUM][6];
int path[100][2];
int src;
int ttl;
int target[TGT+1];
int ttlarray[20];

int parent;
int pn=0;
float ttlavg;
int       cost,       hops,       thops,
minhops,maxhops,mincost,maxcost,minttl,ma
xttl;
int succ;

int found;
int pushed=0;
int idx=0;

int sp=0;
int stack[MAXSTACK][2];

/* Stack Implementation */

void push(int x,int t) {
   if (sp<MAXSTACK) {
      stack[sp][0]=x;
      stack[sp][1]=t;
      sp++;
      pushed=1;
   }
   else {
      printf("Stack Overflow!\n");
      exit(1);
   }
}

void pop(int *x, int *t) {
   if (sp>0) {
      --sp;
      *x=stack[sp][0];
      *t=stack[sp][1];
   }
}

int empty () {
   if (sp<=0)
      return 1;
   else
      return 0;
}

void print_stack() {
   int i, c;

   cost=0; i=0;
   if (empty()==1)
      printf("Stack Empty\n");
   else {
      printf("SP = %d: ",sp);
   for(i=0;i<sp;i++) {

c=chkcost(stack[i][0],stack[i+1][0]);
         printf("%d ",stack[i][0]);

         if (c>0)
            cost+=c;
   }
   }
}

void print_path() {
   int i, c;

   cost=0; hops=0; i=0;
   for(i=0;i<idx;i++) {
         printf("%d ",path[i][0]);

c=chkcost(path[i][0],path[i+1][0]);
         if (c>0)
            cost+=c;
         hops++;
   }
}

int chkcost(int x, int y) {
   int i;
   for (i=0;i<3;i++) {
      if (g[x][i*2]==y)
         return g[x][i*2+1];
   }
   return -1;
}

int chktgt(int node, int ans[3]) {
   int i,j, found;

   found=0;
   for (i=0; i<3; i++) {
      ans[i]=-1;
      if     ((g[node][i*2+1]!=0)     &&
(g[node][i*2]!=parent))
         thops++;
      for (j=0; j<TGT; j++)
      {
         if   ((g[node][i*2+1]!=0)   &&
(g[node][i*2]==target[j])) {
            ans[i]=g[node][i*2];
            found=1;
         }
      }
   }
   return found;
}


int visited(int node) {
   int i;

   for(i=0;i<idx;i++) {
      if (node==path[i][0])
         return 1;
   }
   return 0;
}

void stat() {
   int i;
   int total=0;

   for (i=0;i<ttl;i++)
      total+=ttlarray[i];
   if (pn!=0)
      ttlavg=total/pn;
   else
      ttlavg=0;
}

void chkminmax(int *min, int *max, int x) {
   if (*min==0)
      *min=x;
   else if (*min>x)
      *min=x;

   if (*max==0)
      *max=x;
   else if (*max<x)
      *max=x;
}

/* Time based walker searching */

void tbw(int s, int t) {
   int n,i,flag;
   int ans[3];

   thops=0; pn=0;
   path[idx][0]=s;
   path[idx][1]=t;
   idx++;

   if (chktgt(s,ans)==1) {
      i=0;
      while (i<3) {
         if (ans[i]!=-1) {
            printf("\nTTL=      %d\nPath
Found: ",t);
            print_path();
```

```
            printf("%d",ans[i]);
            cost+=g[s][i*2+1];
            hops++; pn++;
            ttlarray[t]++;

chkminmax(&minhops,&maxhops,hops);

chkminmax(&mincost,&maxcost,cost);

chkminmax(&minttl,&maxttl,t);
            printf("\nCost = %d\tHops =
%d\tMax TTL Left= %d\n",cost,hops,maxttl);
         }
      i++;
      }
   }
   else {
         t--; parent=s;
         if (t>0) {
            if (g[s][5]!=0) {
               push(g[s][4],t);
            }
            if (g[s][3]!=0) {
               push(g[s][2],t);
            }
            if (g[s][1]!=0) {
               push(g[s][0],t);
            }
         }
   }


   while ((empty()==0)) {
      pop(&n,&t);

      i=idx-1; flag=0;
      if (i>0) {
         for (;i>0;i--) {
            if (path[i][1]==t) {
               flag=1;
               break;
            }
         }
      }
      if (flag==1) {
         path[i][0]=n;
         path[i][1]=t;
         idx=++i;
      }
      else {
         path[idx][0]=n;
         path[idx][1]=t;
         idx++;
      }

      if (chktgt(n,ans)==1) {
         i=0;
         while (i<3) {
            if (ans[i]!=-1) {
               printf("\nTTL = %d\nPath
Found: ",t);
               print_path();
               printf("%d ",ans[i]);
               cost+=g[n][i*2+1];
               hops++; pn++;
               ttlarray[t]++;

chkminmax(&minhops,&maxhops,hops);

chkminmax(&mincost,&maxcost,cost);

chkminmax(&minttl,&maxttl,t);
               printf("\nCost = %d\tHops
=        %d\tMax        TTL        Left=
%d\n",cost,hops,maxttl);
               print_path();
            }
            i++;
          }
       }
       else {

            if (pushed==0) {
            for (i=0;i<idx;i++)
               if (path[i][0]==n)
                  parent=path[i-1][0];
            }

            pushed=0; t--;
            if (t>1) {
            if       (g[n][5]!=0        &&
g[n][4]!=parent)
               push(g[n][4],t);
            if       (g[n][3]!=0        &&
g[n][2]!=parent)
               push(g[n][2],t);
            if       (g[n][1]!=0        &&
g[n][0]!=parent)
               push(g[n][0],t);
            if (pushed==1)
               parent=n;
            }
         }
   } /* end of while */

} /* end of tbw */

void read_graph(void) {
   FILE *fp;
   int c,q,i,j,k;
   int n;
   int p=0;
   char buf[20];

   if ((fp =fopen("ts600-0.txt","r")) !=
NULL)
   {
      for(i=0;i<NODE_NUM;i++)
         for(j=0;j<6;j++)
         {
            c=fgetc(fp);
            if (c==EOF)
               break;
            else
            {
               while (c!=EOF && c!=' ' &&
c!='\n')
               {
                  buf[p++]=c;
                  c=fgetc(fp);
               }
               q=0, n=0;
               while (q!=p)

n=n+pow(10,p-q-1)*(buf[q++]-48);
               p=0;
               g[i][j]=n;
            }
         }
   }
   fclose(fp);
   printf("\nThe Graph is\n");
   for(i=0;i<NODE_NUM;i++) {
      printf("Node %d: ",i);
      for(j=0;j<6;j++)
         printf("%d ",g[i][j]);
      printf("\n");
   }
}

void init(int x) {
   int i,n;
   hops=0; thops=0; cost=0;
   minhops=0; maxhops=0;
   mincost=0; maxcost=0;
   minttl=0; maxttl=0;

   for (i=0;i<ttl;i++)
      ttlarray[i]=0;

   RandomInitialise(1803+x,9373-x);
   for (i=0;i<TGT+1;i++) {
         n=RandomInt(0,599);
      target[i]=n;
   }
   src=target[TGT];
}

int main(void) {
   int i,j;
   float  tavgpn,  tavgcost,  tavghops,
tavgthops, tavgttl,rate;

   FILE *fp;
```

```
   fp=fopen("result600.txt","w");
   read_graph();

   for (ttl=1; ttl<=MAXTTL;ttl++) {
      succ=0;
      printf("** TTL = %d **\n",ttl);
      fprintf(fp,"**     TTL     =     %d
**\n\n",ttl);

tavgpn=tavgcost=tavghops=tavgthops=tavgtt
l=rate=0;

      for(i=1;i<SN;i++) {
         init(((i+ttl)*100)%89);


         printf("Simulation %d:\n",i);
         printf("Source Node %d\n",src);
         printf("Target Nodes are: ");


         fprintf(fp,"Simulation
%d:\n",i);
         fprintf(fp,"Source         Node
%d\n",src);
         fprintf(fp,"Target  Nodes  are:
");

         for(j=0;j<TGT;j++) {
            printf("%d ",target[j]);
            fprintf(fp,"%d ",target[j]);
         }

         printf("\n");
         fprintf(fp,"\n");

         thops=0; pn=0;
         tbw(src, ttl);
         stat();

         printf("Max TTL Left = %d\tMin
Cost      =      %d\tMax      Cost      =
%d\n",maxttl,mincost,maxcost);
         printf("Min Hops = %d\tMax Hops =
%d\n",minhops,maxhops);
         printf("Total Paths = %d\tTotal
Hops     =     %d\tAverage     TTL     =
%.2f\n\n",pn,thops,ttlavg);
         fprintf(fp,"Max   TTL   Left   =
%d\tMin   Cost   =   %d\tMax   Cost   =
%d\n",maxttl,mincost,maxcost);
         fprintf(fp,"Min Hops = %d\tMax
Hops = %d\n",minhops,maxhops);
         fprintf(fp,"Total    Paths    =
%d\tTotal  Hops  =  %d\tAverage  TTL  =
%.2f\n\n",pn,thops,ttlavg);
         tavgthops+=thops;
         if (pn!=0) {
            succ++;
            tavgpn+=pn;
tavgcost+=mincost;
            tavghops+=minhops;
tavgttl+=maxttl;
         }
      }
      tavgpn/=SN;          tavgcost/=SN;
tavghops/=SN; tavgthops/=SN;
      tavgttl/=SN;  rate=(float)  succ/
(float) SN;
      printf("TTL = %d Total Number of
Simulation: %d\n",ttl,SN);
      printf("Total Number of Success:
%d\nSuccess Rate=%.2f\n",succ,rate);
      printf("Avg. Paths Found = %.2f\tAvg.
Max TTL Left = %.2f\n",tavgpn,tavgttl);
      printf("Avg. Min Hops = %.2f\tAvg.
Total  Hops  =  %.2f\tAvg.  Min  Cost  =
%.2f\n\n",tavghops,tavgthops,tavgcost);
      fprintf(fp,"TTL = %d Total Number of
Simulation: %d\n",ttl,SN);
      fprintf(fp,"Total Number of Success:
%d\nSuccess Rate=%.2f\n",succ,rate);
      fprintf(fp,"Avg.  Paths  Found  =
%.2f\tAvg.     Max     TTL     Left     =
%.2f\n",tavgpn,tavgttl);
      fprintf(fp,"Avg.   Min   Hops   =
%.2f\tAvg. Total Hops = %.2f\tAvg. Min Cost
= %.2f\n\n",tavghops,tavgthops,tavgcost);
      }
      fclose(fp);
      printf("Finished\n");
      return 0;
};
```