CS5193 Topics in Network

Class Project

Broadcast Sequencing for Efficient Path Query Processing On Network Systems Using Graph Partition Techniques

Submitted to:

Dr. K. Thulasiraman

Submitted by

Jianting Zhang

(000257865)

jianitng@ou.edu

April 25, 2002
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Formalizing the Problem</td>
<td>4</td>
</tr>
<tr>
<td>The Proposed Solution</td>
<td>6</td>
</tr>
<tr>
<td>Implementation Details</td>
<td>7</td>
</tr>
<tr>
<td>Experiments and Results</td>
<td>10</td>
</tr>
<tr>
<td>Related Work</td>
<td>12</td>
</tr>
<tr>
<td>Future Work Directions</td>
<td>13</td>
</tr>
<tr>
<td>References</td>
<td>13</td>
</tr>
<tr>
<td>Appendix: Programs</td>
<td>15</td>
</tr>
<tr>
<td>GenEAFN.cpp</td>
<td>15</td>
</tr>
<tr>
<td>GenPAFN.cpp</td>
<td>21</td>
</tr>
<tr>
<td>GenSeq.cpp</td>
<td>25</td>
</tr>
<tr>
<td>Evaluate.cpp</td>
<td>30</td>
</tr>
</tbody>
</table>
Introduction

Path Query (PQ) is one of the most frequently used functions in network related operations. The number of possible paths in a network is exponential with respect to the number of vertices. It is often not practical storing all the possible paths in a network even in a huge volume disk-based storage system not to say in a wireless broadcast channel with very limited bandwidth.

An alternative is not to retrieve a complete path by just one lookup. Instead, path information is stored inside each node in a compact encoding manner (Agrawal, 1989). Each node (i) has a table with each row a pair of (j,v) where j is the destination and v is the next hop (via) node. Distances from node i to node j, such as shortest path and longest path, can be associated with such a (j,v) pair for query processing. Taking the simple network in Fig. 1 for example, the encoding structure might looks like tables in the right hand of Fig. 1 (each row has four elements that are destination node, via node, critical path distance, shortest distance).

![Fig. 1 Compact Path Encoding Structure](image)

This scheme reduces the storage requirements to $O(b*n^2)$ where $b$ is the branching factor (out-degree) of a node. In this case, path query processing is actually traveling a link list. There is no real time computation cost and no node sequence duplication either in this case.

This scheme is very efficient if all nodes and their associated pointers can be fit into memory. However, this is often not the case when the number of nodes is big due to the $O(b*n^2)$ memory requirement. A solution is to partition the network into several smaller ones so that the collection of tables for the sub-network (we call it a block) is small enough to fit into memory. The whole idea is similar the to conventional page management scheme in an operating system. Originally, they are stored on disks and are dynamically loaded into memory when needed. However, some of the blocks might need to be fanned out in order to accommodate new ones when the nodes along a path are dynamically accessed. If the network is properly partitioned, the nodes along many paths fall into one or a few blocks and the total number of fan-in/fan-out is reduced. An optimal solution to partition the graph minimizing the number of fan-in/fan-out during path query process is desirable.
To the best of knowledge, none of existing work has considered using broadcast channel as data access media for network data broadcasting. It is well known that broadcasting has excellent scalability – its performance is independent of number of users. Free of mobility restrictions within broadcast range is another good feature. These are really desirable features in many wireless mobile network related applications, real time traffic guide in urban area for example. Unlike memory or disk based data access that can use random access method, data in broadcast channel is one-dimensional sequential in nature. Besides the number of blocks to be accessed (Tune In Time – TT), the duration between the time starting accessing broadcast channel to the time of all data are retrieved (Access Time or Latency – AT) is another important parameter in evaluating broadcast system performance (Imielinski, 1997). Since tuning to a data block in active mode is much more power consuming than staying in sleeping mode, TT is more important than AT. In data broadcast researches, the tradeoff between TT and AT is also well-known (Imielinski, 1997).

In the following example (Fig. 2), the whole network is partitioned into three sub-networks and they are stored in block A, B and C respectively and broadcast sequentially. If we want to query a path between node 1 and node 11, then we must tune in to block A and C to retrieve data thus TT is len(A)+len(C). Although our mobile terminal can be in sleeping mode during block B broadcast, our query process can not be finished until the end of block C, thus our AT is len(A)+len(B)+len(C).

Fig. 2 Network Partition and its Broadcasting Sequence

Formalizing the Problem

We are interested in broadcasting a network and its paths (all-pair shortest paths for example) information over wireless broadcast channel. Given source and destination nodes, we want to minimize the TT (measured by bytes of data need to download) and AT (also in terms of bytes of data) in querying its paths over air media. We assume each disk block is corresponding to a broadcast block in a broadcast cycle and a client (mobile user) either download a block as a whole or not download it at all. Thus the optimization
objective in minimizing fan-in/fan-out in disk-based approach is equivalent to minimizing the TT in broadcast scenario.

Assume the network has N nodes and is partitioned into M sub-networks (G_m), each sub-network has K_m nodes, each node has L_n entries of (destination, via, dist). For the sake of simplicity, we assume each entry is of unit size, thus the total size of each sub-network is S_m = \sum_{n=1}^{K_m} L_n, n \in G_m. Assume the total memory at the client side is MEM.

Function SGN (m) is defined to be 1 if block (sub-network) G_m is in the memory and 0 otherwise. Function NET mapping node i to a sub-network and function SIZE mapping node i to the size of a sub-network G_m where it belongs to.( i.e, S_m).

For any given node pair of (i,j), there are P_{ij} nodes along its shortest path with the number of N_{1}^{(i,j)}, N_{2}^{(i,j)}, … N_{P}^{(i,j)}.

\[ \sum_{i=1}^{N} \sum_{j=1}^{N} \left( \sum_{k=1}^{P_{ij}} SGN\left( NET\left( N_{k}^{(i,j)} \right) \right) \right) \times SIZE\left( NET\left( N_{k}^{(i,j)} \right) \right) \]

S.T.
1) S_m<MEM, for any m=1,M
2) Node n only belongs to one particular G_m, for any n=1,N

For access time (AT), we further assume a client start to query the broadcast channel about path P_{ij} at time B_{ij} and End of the query at time E_{ij}, the objective function can be written as:

\[ \sum_{i=1}^{N} \sum_{j=1}^{N} (E_{ij} - B_{ij}) \]

Practically, the begin time of a shortest path query can be any time during a broadcast cycle. For the sake of simplicity, we assume B_{ij}=0 which means we always start a query at the beginning of a broadcast cycle, thus the objective function can be further written as:

\[ \sum_{i=1}^{N} \sum_{j=1}^{N} E_{ij} \]

The reason behind zero beginning assumption is that it makes result deterministic given a network and its partitions while the random start case will make results incomparable unless a great number of tests are performed before the average converges.
The Proposed Solution

The key idea behind our proposed solution is to construct new networks based on access frequencies of old network. Then we partition the constructed network into desired sub-networks by successive bisections. Partition refinement technique is used to achieve optimality and meet size balance requirement as well.

The solution consists of two steps. The first step is to minimize tune in time. We count the access frequencies for each edge in the original network and construct a network take these frequencies as the distances (fig. 3). We call the derived network as Edge Access Frequency Network (EAFN). EAFN has the same topology as the original network and the only difference is the distances associated with them. Our heuristic in this step is based on the observation that minimizing the cuts is the same as minimizing inter-block communication costs which is exactly our goal of minimizing TT.

Fig. 3 Deriving EAFN by Summarizing Edge Access Frequencies

Since a block is expected to be downloaded by the clients as a whole and can be accessed randomly within the block, the sequences of nodes within a block is insignificant. However, the sequence of partitioned blocks will affect the access time (latency) significantly. To handle this problem, we developed another heuristic also based on graph partition in the second step.

In this step, we first count on the lengths of paths with source in a block and destination in another block based on the block partition result in the first step. We then construct a network by attaching edges of source-destination pairs in the original network that take these numbers as the distances associated with them. We call the constructed network Path Access Frequency Network (PAFN). Note that PAFN is likely to be dense. A decomposition tree is then obtained by successive bisections of the PAFN. We finally can obtain a block ordering sequence by pre-order (or other orders) traveling the leaves of the decomposition tree. In this step, we try to maximize the cuts rather than minimize the cuts in the first step. The heuristic is based on the observation that spreading blocks
whose nodes have longer shortest paths far away with each other will leave the room for blocks whose nodes have shorter shortest paths to be close to each other and thus reduce total/average access time.

![Diagram of a graph with nodes and shortest paths](image)

**Fig. 4 Deriving PAFN by Summarizing Block Access Frequencies**

### Implementation Details

We use a public available software package for graph partitioning, METIS 4.0 ([HREF 1](#)). The most popular type of path query, the shortest path query is used as an example for performance evaluation. The general workflow of the proposed solution is shown in Fig. 5.

We have developed four auxiliary modules for pre-processing and post-processing in order to obtain a broadcast sequence and perform evaluations on it.

The first module, GenEAFN, takes node link list as input, compute all-pair shortest paths, calculate edge access frequencies and generate original network (ORGN) as well as EAFN in METIS format. Several intermediate matrices are stored in corresponding files to be used in the following two modules. The first one is ORGN shortest distance matrix (saved as “.distance”) which has the shortest distance values for all source-destination pairs. The second is ORGN shortest path matrix (saved as “.path”) where p[i][j]=k means the shortest path between source i and destination j goes through intermediate node k. By recursively calling a “path” function (described below) we can find all the intermediate nodes along the shortest path between a source-destination pair. The third matrix is the ORGN shortest path length (in terms of number of edges, saved as “.length”) which is used to generate PAFN. This matrix is also obtained in the “path” function.

The second module, GenPAFN, takes EAFN, the METIS partition result (EAFN.graph.part.* where * is the number of partitions) and the ORGN shortest path length file as inputs and generates PAFN.

The third module, GenSeq, takes sequencing file of PAFN (either by graph partition or other methods), the raw ORGN data file, ORGN distance file, ORGN path
file and EAFN partition result file as inputs and finally generates sequence file of original network.

---

**Fig. 5 Implementation Procedure**

The fourth module is for evaluation. Given a broadcast partitioned sequence and a query path, this module will calculate total energy consumption for downloading the path from the broadcast sequence in terms of tune in time and access time. In this study we also calculate the total/average energy consumption for all pair shortest paths. For comparison purposes, we also include several functions for generating random sequencings.

We next explain several key functions used in this study. First of all, the **recursive shortest path discovery function** plays an important role in generating PAFN and final broadcast sequence. It is briefly described as follows:

Input: source node u, destination v, path matrix p (where p[i][j]=k; matrix p is obtained form Floyd-Warshall algorithm)
Output: counting edge access frequencies along the shortest path and return its length

Procedure:
  \( k = p[u][v] \);
  if \( k = -1 \)
    u and v are directly connected, return 1;
  if \( u = k \) or \( k = v \)
    print error message (this can not happen)
  if \( p[u][k] = -1 \)
    (* u and k are directly connected and no SP passes through it *)
    increase access frequency \( w[u][k] \) by 1
    set the length of SP between (u,k) \( p1 \) to 1.
  Else
    set \( p1 \) to path(u,k);
  Processing (k,v) similarly and set \( p2 \).
  return \( p1 + p2 \);

Second, the procedure for **generating final broadcast sequence** is as follows:

Input:
- block sequence array \( b \)
- node partition array \( r \)
- ORGN distance matrix \( c \)
- ORGN SP distance matrix \( d \)
- ORGN path matrix \( p \)

Output: Broadcast sequence

Procedure:
  Set up an array \( \text{cnt}[\text{numBlock}] \) to store number of nodes in each block, array \( \text{ss}[\text{numBlock}] \) to store broadcast sequence in each block
  For each source node \( i \)
    Increase \( \text{cnt}[r[i]] \) by 1;
    Output node id to \( \text{ss}[r[i]] \);
  For each destination node \( j \)
    If \( (i == j) \) continue;
    Set “to „ to \( j + 1 \);
    If \( (c[i][j] != 0) \) //iI and j are connected directly
      Set “via” to \( j + 1 \) //via itself
      Set “dist” to \( c[i][j] \)
    Else
      Set “via” to \( p[i][j] + 1 \); //node numbered start from 1
      Set “dist” to \( d[i][j] \); //derived from SP calculation
  End
  Output “to,via,dist” pair to \( \text{ss}[r[i]] \);
  End
  For each block
  Output \( \text{cnt}[b[i] - 1] \) to sequence file; //numbered start with 1
Output ss[b[i]-1] to sequence file;

Third, the procedure for **evaluating a path** is as follows:

**Input:**
- block sequence array b
- Number of nodes in each block array n
- node partition array r
- path array p

**Output:** tune in time and access time for a path query over a broadcast sequence

**Procedure:**
- Set up an array (cnt) for holding all the visited blocks (initially empty)
- For each node in the path p[i]
  - Set the visiting block id to b[r[p[i]]]
  - If the id has not been found in cnt, append it to the end of cnt
- Find the minimum sequence index (minind) to minimum value of sequence indices of all the blocks in cnt; similarly for maxind.
- Set tune in time to the number of ids in cnt
- Set access time to the summation of the numbers of nodes in the block whose sequence index is between minind and maxind.

**Experiments and Results**

We use a simple network (Fig. 6) to demonstrate our ideas. Its 3-part METIS partition is also shown in the figure for comparison purpose. The derived EAFN and its 3-part METIS partition is shown in Fig 7. The derived PAFN and its decomposition tree is shown in Fig. 8. We obtained the tree by inspection science it is a very small network.

We report our results in terms of unit for all the $12 \times (12-1) = 132$ pairs of shortest paths. The length of each unit is the products of $n_1$, $n_2$ and $n_3$. Where $n_1$ is one less the number of nodes since the shortest path of a node to its self is trival. $n_2$ is the number of items in a row in the compact path encoding structure (it is 3 for our (destination, via, dist) pair). $n_3$ is the size of each data item (1/2/4 bytes).

The results of our proposed solution on the network used in the experiment are 238 and 1040 units for tune in time and access time respectively. We perform this method on the random sequencings and block segmentations on the same network for 1000 times. The average tune in time is 276.334 units and the average access time is 1297.218 units. The results are 16.1% and 24.7% higher than our results respectively.

Among the 1000 runs, 60 of them have equal block sizes (each block has 4 nodes in the 3 blocks). There average tune in time is 295.833 units and the average access time is 1283.6 units that are 24.3% and 23.4% higher than our results. Since equal size partition is very important in efficient data access and tune in time is more important than access time, we draw the conclusion that our method has much better performance than random broadcast sequencing. It also suggests that our heuristics of constructing EAFN and PAFN are effective.
Fig. 6 A Network (ORGN) Used in the Experiment and its 3 parts METIS Partition

Fig. 7 Derived EAFN and its 3 parts METIS Partition
We have also evaluated our proposed method on a real transportation network of the state of Taxas (Fig. 8). It has 62 nodes (cities) and 120 edges (major roads). We partition the derived EAFN into eight parts using METIS. They are again sequenced based on the hierarchical decomposition tree. Note the tree is not obtained from METIS because METIS outputs do not give sequence information. Instead, we write our own optimal bi-section routine. Since the number of blocks is much smaller than the number of nodes, we can afford searching the maximum cuts of all possible combinations (with the restriction of equal division) in each level of tree decomposition.

For the (120*119=14280) pairs of shortest paths, the tune in time and access time for our results are 11556 and 122932 units respectively. The corresponding results of 1000 runs are 15046.817 and 182692,

Related Work

Although the general graph partitioning problem is known to be NP-hard, there are several good heuristic algorithm for graph portioning. Two comprehensive surveys of graph partitioning are available, one from netlist partitioning for VLSI CAD perspective (Alpert 1995) and the other one from the viewpoint of efficient execution of scientific simulations on high performance parallel computers (Schloegel, 2000). The Kernighan – Lin (Kernighan, 1970) and Fiduccia-Mattheyses (Fiduccia,1982) are two well-known graph partition refinement algorithms. FM algorithm is used in METIS.
Sequencing graph nodes can be treated as a graph layout problem. A survey on graph layout problem is presented in (Daíz, 2002). Spreading metric algorithms are proposed to solve a special type of graph layout problem known as Linear Arrangement (ordering) problem which is similar to our application context (Even, 2000; Rao, 1998; Kuo, 1997).

Compact path encoding structure is first proposed in (Agrawal, 1989) and later extended to hierarchical scheme for path query in transportation networks by (Jing, 1998). (Larson, 1989) proposed a file structure supporting traversal recursion. Materialization trade-offs in hierarchical shortest path algorithms is addressed in (Shekhar, 1997). Finding shortest path in large networks based on hierarchical graphs are proposed in (Zhan, 2001).

**Future Work Directions**

We have developed two heuristics in constructing EAFN and PAFN. Experiment results on broadcast sequencing based on graph partition results on EAFN and PAFN have shown that they have significant improvements over random sequencings.

The theoretical perspective of future work is to investigate the possibility of integrating spread metric based Minimum Linear Arrangement (minLA) methods with local edge connectivity based partition refinement methods for graph partition and sequencing. The idea is to work on original network directly and minimize tune in time and access time of broadcast sequence simultaneously by constructing spread metric using linear programming methods.

The practical perspective of future work is to develop our own graph partition software package that integrate the heuristics developed in this study as well as the ideas stated above. We will evaluate the performances of the proposed method in this paper as well as possible new methods in the future on some other networks.

**References**


Edward P.F. Chan and Ning Zhang, Finding Shortest Paths in Large Network Systems, ACM-GIS 2001,


K. Schloegel and G. Karypis and V. Kumar, Graph Partitioning for High Performance Scientific Simulations, In J. Dongarra et al., editor, CRPC Parallel Computing Handbook. Morgan Kaufmann, 2000


Appendix: Programs

GenEAFN.cpp
/*************************************************************************/
CS5913 Class Project
GenEAFN.cpp

Description: module for generating Edge Access Frequency
Netowork(EAFN) to be used in METIS for graph partition.

Jianting Zhang 04/25/02
*************************************************************************/

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

int numNode,numEdge,numType=1;//default network type for metis
int **d;//work matrix for shorest path calculation
int **p;//p[i][j]=k, which means i goes to j through k
int **w;//edge access frequency matrix
int **l;//l[i][j]=len(sp(i,j))

int path(int u,int v)
{
    int p1=0,p2=0;
    int k=p[u][v];
    if(k==-1) // u and v are directly connected
        return(1);
    if(u==k||k==v) printf("waring...%5d %5d %5d
",u,k,v);
    //u and k are directly connected and no SP passes through it,increase access frequency
    if(p[u][k]==-1)
    {
        w[u][k]++;
        p1=1;
    }
    else p1=path(u,k);
    //u and k are directly connected and no SP passes through it,increase access frequency
    if(p[k][v]==-1)
    {
        w[k][v]++;
        p2=1;
    }
    else
    {
        p2=path(k,v);
        return(p1+p2);
    }
}

void main()
{
    FILE *fp;
    char filename[30];
    char ORGNRoot[10]="ORGN";
char EAFNRoot[10]="EAFN";
    char PAFNRoot[10]="PAFN";
    char dataExt[10]=".data";
char graphExt[10]=".graph";
    char pathExt[10]=".path";
    char distanceExt[10]=".distance";
    char lengthExt[10]=".length";
    int i,j,k,count=0;

    strcpy(filename,ORGNNRoot);
    strcat(filename,dataExt);
    if(( fp=fopen(filename,"r"))==NULL)
    {
        printf("can not open data file %s\n",filename);
        exit(-1);
    }
    fscanf(fp," %d ",&numNode);
    printf("%10d\n",numNode);
    d=new int *[numNode];
    p=new int *[numNode];
    w=new int *[numNode];
    l=new int *[numNode];
    for(i=0;i<numNode;i++)
    {
        d[i]=new int[numNode];
        p[i]=new int[numNode];
        w[i]=new int[numNode];
        l[i]=new int[numNode];
        for(j=0;j<numNode;j++)
        {
            d[i][j]=0;
            p[i][j]=-1;
            w[i][j]=0;
        }
        d[i][i]=0;
    }
    numEdge=0;
    for(i=0;i<numNode;i++)
    {
        int nums,to,weight;
        fscanf(fp," %d ",&nums);
        printf("%10d\n",nums);
        numEdge+=nums;
        for(j=0;j<nums;j++)
        {
            fscanf(fp," %d %d ",&to,&weight);
            printf("%5d %5d",to,weight);
            d[i][to-1]=weight;
        }
        printf("\n");
    }
    fclose(fp);

    printf("now printing original network distance matrix.....\n");
    for(i=0;i<numNode;i++)
    {
for(j=0;j<numNode;j++)
{
    char c;
    if (d[i][j]!=d[j][i]) c='*';
    else c='';
    printf("%5d%c",d[i][j],c);
}
printf("n");
}
if(numEdge%2!=0)
{
    printf("num edge not even!!!!");
    exit(-1);
}
numEdge/=2;

/* writing out original network file in METIS format */
strcpy(filename,ORGNRoot);
strcat(filename,graphExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open graph file %s\n",filename);
    exit(-1);
}
fprintf(fp,"%10d %10d %10d\n",numNode,numEdge,numType);
for(i=0;i<numNode;i++)
{
    int count=0;
    for(j=0;j<numNode;j++)
        if(d[i][j]>0)
        {
            fprintf(fp,"%5d %5d",j+1,d[i][j]);
            count++;
        }
    if(count>0) fprintf(fp,"n");
}
fclose(fp);

/* calculating all pair shortest paths using Floyd-Warshall algorithm*/
for(i=0;i<numNode;i++)
    for(j=0;j<numNode;j++)
        if(d[i][j]==0) d[i][j]=10000;
for(k=0;k<numNode;k++)
    for(i=0;i<numNode;i++)
        for(j=0;j<numNode;j++)
        {
            int td=d[i][k]+d[k][j];
            if(td<d[i][j])
            {
                d[i][j]=td;
                p[i][j]=k;
            }
        }
printf("now printing original network SP distance matrix.....\n");
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++)
    {
        if(d[i][j]>1000) d[i][j]=0;
        char c;
        if (d[i][j]!=d[j][i]) c='*';
        else c='.';
        printf("%5d%c",d[i][j],c);
        printf("\n");
    }
    printf("\n");
}

printf("now printing original network path matrix.....\n");
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++)
    {
        char c;
        if (p[i][j]!=p[j][i]) c='*';
        else c='.';
        printf("%5d%c",p[i][j],c);
        printf("\n");
    }
    printf("\n");
}/* summerizing access frequencies for each edge and SP length of a s-t pair*/
for(i=0;i<numNode;i++)
    for(j=0;j<numNode;j++)
    {
        l[i][j]=path(i,j);
    }

printf("now printing edge access frequence matrix.....\n");
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++)
    {
        char c;
        if (w[i][j]!=w[j][i]) c='*';
        else c='.';
        printf("%5d%c",w[i][j],c);
        printf("\n");
    }
    printf("\n");
}
/* generating EAFN */
numEdge=0;
for(i=0;i<numNode;i++)
    for(j=0;j<numNode;j++)
    {
        if(w[i][j]>0) numEdge++;
    }
if(numEdge%2!=0)
{
    printf("num edge not even!!!!\n");
}
exit(-1);
}
numEdge/=2;

strcpy(filename,EAFNRoot);
strcat(filename,graphExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open file for output:%s\n",filename);
    exit(-1);
}
fprintf(fp,"%5d %5d %5d\n",numNode,numEdge,numType);
for(i=0;i<numNode;i++)
{
    int count=0;
    for(j=0;j<numNode;j++)
        if(w[i][j]>0)
        {
            fprintf(fp,"%5d %5d ",j+1,w[i][j]);
            count++;
        }
    if(count>0) fprintf(fp,"\n");
}
fclose(fp);

/* writing out path information to be used for generating broadcast sequence later*/
strcpy(filename,ORGNRoot);
strcat(filename,pathExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open path file for output:%s\n",filename);
    exit(-1);
}
fprintf(fp,"%5d\n",numNode);
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++) fprintf(fp,"%5d ",p[i][j]);
    fprintf(fp,"\n");
}
fclose(fp);

/* writing out distance information to be used for generating broadcast sequence later*/
strcpy(filename,ORGNRoot);
strcat(filename,distanceExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open distance file for output:%s\n",filename);
    exit(-1);
}
fprintf(fp,"%5d\n",numNode);
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++) fprintf(fp,"%5d ",d[i][j]);
}
fp = fopen(fp,"n");
}
close(fp);

/* writing out distance information to be used for generating PAFN*/
strcpy(filename,ORGNRoot);
strcat(filename,lengthExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open distance file for output:%s\n",filename);
    exit(-1);
}
fp = fprintf(fp,"%5d\n",numNode);
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++) fprintf(fp,"%5d ",l[i][j]);
    fprintf(fp,"\n");
}
close(fp);

/* clean up */
for(i=0;i<numNode;i++)
{
    delete d[i];
    delete p[i];
    delete w[i];
    delete l[i];
}
delete d;
delete p;
delete w;
delete l;
GenPAFN.cpp

CS5913 Class Project
GenPAFN.cpp

Description: module for generating Page Access Frequency Network (PAFN) for block sequencing.

Jianting Zhang 04/25/02
*********************************************************/

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

int numNode, numEdge=0, numType=1, numBlock;
int *r; // block number a node belongs to
int **l; // PAFN path length information
int **c; // PAFN edge distance

void main(int argc, char** argv)
{
  FILE *fp;
  char filename[30];
  char ORGNRoot[10]="ORGN";
  char EAFNRoot[10]="EAFN";
  char PAFNRoot[10]="PAFN";
  char dataExt[10]=".data";
  char graphExt[10]=".graph";
  char pathExt[10]=".path";
  char distanceExt[10]=".distance";
  char lengthExt[10]=".length";
  char partExt[20]=".graph.part.3";
  int i, j;

  if(argc!=3)
  {
    printf("wrong parameters: USAGE\n");
    printf("GenPAFN numNode numBlock\n");
    exit(-1);
  }

  numNode=atoi(argv[1]);
  numBlock=atoi(argv[2]);
  if((numNode<=0)||(numBlock<=0))
  {
    printf("wrong node num or block num\n");
    exit(-1);
  }

  r = new int[numNode];
  c = new int *[numBlock];
  l = new int *[numNode];
  for(i=0; i<numBlock; i++)
  {
    c[i] = new int[numNode];
  }
  for(i=0; i<numNode; i++)
  {
    l[i] = new int[numBlock];
  }

  printf("numNode\n");
  for(i=0; i<numNode; i++)
    printf("%d, ", r[i]);
  printf("\n");

  printf("numBlock\n");
  for(i=0; i<numBlock; i++)
    printf("%d, ", c[i][0]);
  printf("\n");

  printf("l\n");
  for(i=0; i<numNode; i++)
  {
    for(j=0; j<numBlock; j++)
      printf("%d, ", l[i][j]);
    printf("\n");
  }

  printf("c\n");
  for(i=0; i<numBlock; i++)
  {
    for(j=0; j<numNode; j++)
      printf("%d, ", c[i][j]);
    printf("\n");
  }

  printf("r\n");
  for(i=0; i<numNode; i++)
    printf("%d, ", r[i]);
  printf("\n");

  printf("\n");

  fp = fopen(filename, "w");
  fprintf(fp, "%d\n", numNode);
  fprintf(fp, "%d\n", numBlock);
  for(i=0; i<numNode; i++)
    fprintf(fp, "%d\n", r[i]);
  for(i=0; i<numBlock; i++)
    fprintf(fp, "%d\n", c[i][0]);
  for(i=0; i<numNode; i++)
  {
    for(j=0; j<numBlock; j++)
      fprintf(fp, "%d\n", l[i][j]);
  }
  for(i=0; i<numBlock; i++)
  {
    for(j=0; j<numNode; j++)
      fprintf(fp, "%d\n", c[i][j]);
  }
  for(i=0; i<numNode; i++)
    fprintf(fp, "%d\n", r[i]);

  fclose(fp);
}
{  
c[i]=new int[numBlock];
  for(j=0;j<numBlock;j++) c[i][j]=0;
}
for(i=0;i<numNode;i++)
{
  l[i]=new int[numNode];
  for(j=0;j<numNode;j++) l[i][j]=0;
}

/* reading block partition result*/
strcpy(filename,EAFNRoot);
strcat(filename,partExt);
if(( fp=fopen(filename,"r"))==NULL)
{
  printf("can not open data file: %s\n",filename);
  exit(-1);
}
for(i=0;i<numNode;i++)
{
  int temp;
  fscanf(fp,"%d",&temp);
  r[i]=temp;
}
fclose(fp);

strcpy(filename,ORGNRoot);
strcat(filename,lengthExt);
if(( fp=fopen(filename,"r"))==NULL)
{
  printf("can not open data file: %s\n",filename);
  exit(-1);
}

int pn;
scanf(fp,"%d ",&pn);

for(i=0;i<numNode;i++)
{
  int temp;
  for(j=0;j<numNode;j++)
  {
    fscanf(fp,"%d ",&temp);
    l[i][j]=temp;
  }
}
fclose(fp);

printf("now printing length matrix.....\n");
char s;
for(i=0;i<numNode;i++)
{
  for(j=0;j<numNode;j++)
  {
    if(l[i][j]!=l[j][i]) s='*';
  }
}
else s=' ';
    printf("%5d%c",l[i][j],s);
}
printf("n");
}

for(i=0;i<numNode;i++)
    for(j=0;j<numNode;j++)
        if(l[i][j]>0) c[r[i]][r[j]]+=l[i][j];

int maxCut=0;
for(i=0;i<numBlock;i++)
    for(j=0;j<numBlock;j++)
    {
        if(i==j) continue;
        if(c[i][j]>maxCut) maxCut=c[i][j];
        if(c[i][j]>0) numEdge+;
    }
printf("now printing PAFN distance matrix.....\n");
for(i=0;i<numBlock;i++)
    {
        for(j=0;j<numBlock;j++) printf("%5d",c[i][j]);
        printf("n");
    }

if(numEdge%2!=0)
    {
        printf("num edge not even!!!!");
        exit(-1);
    }
numEdge/=2;
/* generating PAFN dege distances */

strcpy(filename,PAFNRoot);
strcat(filename,graphExt);
if((fp=fopen(filename,"w"))==NULL)
    {
        printf("can not open data file: %s\n",filename);
        exit(-1);
    }
fprintf(fp,"%10d %10d %10d\n",numBlock,numEdge,numType);
for(i=0;i<numBlock;i++)
    {
        int count=0;
        for(j=0;j<numBlock;j++)
        {
            if((i!=j)&&(c[i][j]>0))
            {
                fprintf(fp,"%5d %5d",j+1,c[i][j]);
                count+;
            }
        }
        if(count>0) fprintf(fp,"n");
    }
fclose(fp);
/* clean up */
for(i=0; i<numNode; i++)   delete l[i];
for(i=0; i<numBlock; i++) delete c[i];
delete l;
delete c;
delete r;
}
GenSeq.cpp
/******************************/
CS5913 Class Project
GenSeq.cpp

Description: module for generating final broadcast sequence based on the result of EAFN partition and PAFN partition.

Jianting Zhang 04/25/02
*****************************/

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

int numNode,numEdge,numBlock,numType;
int *b;//block sequence
int *r;//block number a node belongs to
int **c;//original network edge distance
int **p;//path information derived from shortest path computation
int **d;//shortest distances derived from shortest path computation

void main()
{
    FILE *fp;
    char filename[30];
    char ORGNRoot[10]="ORGN";
    char EAFNRoot[10]="EAFN";
    char PAFNRoot[10]="PAFN";
    char dataExt[10]=".data";
    char graphExt[10]=".graph";
    char pathExt[10]=".path";
    char distanceExt[10]=".distance";
    char sequenceExt[10]=".sequence";
    char blockExt[10]=".block";
    char partExt[20]=".graph.part.";

    int i,j;

    /* reading final block sequence*/
    strcpy(filename,PAFNRoot);
    strcat(filename,blockExt);
    if((fp=fopen(filename,"r"))==NULL)
    {
        printf("can not open block file: %s\n",filename);
        exit(-1);
    }

    fscanf(fp,"%d",&numBlock);
    b=new int[numBlock];
    for(i=0;i<numBlock;i++)
        fscanf(fp,"%d",&b[i]);
    fclose(fp);
/* we still need the original network data to know whether two nodes have directly connection*/
strcpy(filename, ORGNRoot);
strcat(filename, dataExt);
if ((fp = fopen(filename, "r")) == NULL)
{
    printf("can not open data file: %s\n", filename);
    exit(-1);
}
scanf(fp, "%d %d %d", &numNode, &numEdge, &numType);
printf("%10d %10d %10d\n", numNode, numEdge, numType);
c = new int *[numNode];
d = new int *[numNode];
p = new int *[numNode];
r = new int [numNode];
for (i = 0; i < numNode; i++)
{
    c[i] = new int[numNode];
    d[i] = new int[numNode];
    p[i] = new int[numNode];
    for (j = 0; j < numNode; j++)
    {
        c[i][j] = 0;
        d[i][j] = 0;
        p[i][j] = 0;
    }
}
for (i = 0; i < numNode; i++)
{
    int nums, to, weight;
    fscanf(fp, "%d ", &nums);
    printf("%10d ", nums);
    for (j = 0; j < nums; j++)
    {
        fscanf(fp, "%d %d", &to, &weight);
        printf("%5d %5d", to, weight);
        c[i][to - 1] = weight;
    }
    printf("\n");
}
fclose(fp);

printf("now printing original network distance matrix.....\n");
for (i = 0; i < numNode; i++)
{
    for (j = 0; j < numNode; j++)
    {
        printf("%5d", c[i][j]);
        printf("\n");
    }
}

/* reading original network SP distance information*/
strcpy(filename, ORGNRoot);
strcat(filename, distanceExt);
if ((fp = fopen(filename, "r")) == NULL)
{
    printf("can not open path file");
}
exit(-1);

int dn;
scanf(fp,"%d ",&dn);
for(i=0;i<numNode;i++)
{
    int temp;
    for(j=0;j<numNode;j++)
    {
        fscanf(fp,"%d ",&temp);
        d[i][j]=temp;
    }
}
fclose(fp);
printf("now printing orginal network SP distance matrix.....\n");
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++) printf("%5d",d[i][j]);
    printf("\n");
}

strcpy(filename,ORGNRoot);
strcat(filename,pathExt);
if((fp=fopen(filename,"r"))==NULL)
{
    printf("can not open path file: %s",filename);
    exit(-1);
}

int pn;
scanf(fp,"%d ",&pn);
for(i=0;i<numNode;i++)
{
    int temp;
    for(j=0;j<numNode;j++)
    {
        fscanf(fp,"%d ",&temp);
        p[i][j]=temp;
    }
}
fclose(fp);

printf("now printing orginal network SP path matrix.....\n");
for(i=0;i<numNode;i++)
{
    for(j=0;j<numNode;j++) printf("%5d",p[i][j]);
    printf("\n");
}

strcpy(filename,EAFNRoot);
strcat(filename,partExt);
char s1[10];
itoa(numBlock,s1,10);
strcat(filename,s1);
if((fp=fopen(filename,"r"))==NULL)
{
    printf("can not open part file for read:%s",filename);
    exit(-1);
}

for(i=0;i<numNode;i++)
{
    int temp;
    fscanf(fp,"%d",&temp);
    r[i]=temp;
}
fclose(fp);

char **ss;
ss=new char* [numBlock];
int *cnt=new int[numBlock];
for(i=0;i<numBlock;i++)
    ss[i]=new char[5000];

for(i=0;i<numBlock;i++)
{
    strcpy(ss[i],"");
    cnt[i]=0;
}

strcpy(filename,ORGNRoot);
strcat(filename,sequenceExt);
if((fp=fopen(filename,"w"))==NULL)
{
    printf("can not open sequence file for output:%s",filename);
    exit(-1);
}

for(i=0;i<numNode;i++)
{
    cnt[r[i]]++;;
    char s[100];
    sprintf(s,"%10d
",i+1);
    strcat(ss[r[i]],s);
    for(j=0;j<numNode;j++)
    {
        if(i==j) continue;
        int to=j+1,via,dist;
        if(c[i][j]!=0)
        {
            via=j+1;
            dist=c[i][j];
        }
        else
        {
            via=p[i][j]+1;
        }
    }
}
dist = d[i][j];
}
sprintf(s, "%5d %5d %5d\n", to, via, dist);
strcat(ss[r[i]], s);
}
}

fprintf(fp, "%10d\n", numNode);
for (i = 0; i < numNode; i++) fprintf(fp, "%5d\n", r[i] + 1);
fprintf(fp, "\n");
for (i = 0; i < numBlock; i++)
{
    fprintf(fp, "%5d\n", cnt[b[i] - 1]); // b[i] states from 1
    fprintf(fp, "%s", ss[b[i] - 1]);
}
fclose(fp);

for (i = 0; i < numBlock; i++) delete ss[i];
for (i = 0; i < numNode; i++)
{
    delete c[i];
    delete d[i];
    delete p[i];
}
delete b;
delete r;
delete c;
delete d;
delete p;
delete ss;
delete cnt;
Evaluate.cpp

/**************************************************************************
CS5913 Class Project
Evaluate.cpp
Description: module for evaluating the performance of a broadcast sequence
by calculating the tune in time (TT) and the access time (AT) for all pair
shortest paths in a network. Some parameters are hard-coded.

Jianting Zhang 04/25/02
**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>

typedef struct
{
    int tt; //tune in time
    int at; //access time
} cost;

typedef struct
{
    int numNode; //number of ndoes in the graph
    int numBlock;//number of partitioned blocks
    int *b; //block segmenation
    int *r; //node partition
    int *n; //number of nodes in each block
} partition;

typedef struct
{
    int numNode; //number of ndoes in the graph
    int **p; //intermediate path matrix
    int **l; //length of all-pair shortest paths
    int ***path;//nodes of all-pair shortest paths in a sequence
} network;

void freePS(partition p)
{
    delete p.b;
    delete p.r;
    delete p.n;
}

void freeNS(network n)
{
    for(int i=0;i<n.numNode;i++)
    {
        for(int j=0;j<n.numNode;j++)
            delete n.path[i][j];
    }
}
delete n.l[i];
delete n.p[i];
delete n.path[i];
}
delete n.path;
delete n.l;
delete n.p;
}

void printPS(partition p)
{
    printf("node sequence\n");
    for(int i=0;i<p.numNode;i++)
        printf("%3d",p.r[i]);
    printf("\n");
    printf("block sequence\n");
    for(i=0;i<p.numBlock;i++)
        printf("%3d/%3d",p.b[i],p.n[i]);
    printf("\n");
}

/* get the path give source u and destination v, the result is stored in path*/
int getPath( int **p,int u,int v,int* path,int maxNode)
{
    int p1,p2;
    int *n1=new int[maxNode];
    int *n2=new int[maxNode];
    int k=p[u][v];
    if(k==-1) // u and v are directly connected
    {
        path[0]=v;
        return(1);
    }
    if(u==k||k==v) printf("waring...%5d %5d %5d\n",u,k,v);
    //u and k are directly connected and no SP passes through it,increase access frequency
    if(p[u][k]==-1)
    {
        p1=1;
        n1[0]=k;
    }
    else
    p1=getPath(p,u,k,n1,maxNode);
    //u and k are directly connected and no SP passes through it,increase access frequency
    if(p[k][v]==-1)
    {
        p2=1;
        n2[0]=v;
    }
    else
    p2=getPath(p,k,v,n2,maxNode);
    for(int i=0;i<p1;i++)
        path[i]=n1[i];
delete n1;
    for(i=0;i<p2;i++)
        path[p1+i]=n2[i];
delete n2;
return(p1+p2);
}

network readNetowrk(char* pathFile, char *lengthFile)
{
    network n;
    FILE *fp;
    int numNode,i,j,temp;

    /* read path file*/
    if((fp=fopen(pathFile,"r"))==NULL)
    {
        printf("can not open data file \%s\n",pathFile);
        exit(-1);
    }
    fscanf(fp," %d",&numNode);
    n.numNode=numNode;
    n.p=new int*[numNode];
    n.l=new int*[numNode];
    n.path=new int**[numNode];
    for(i=0;i<numNode;i++)
    {
        n.path[i]=new int*[numNode];
        n.p[i]=new int[numNode];
        n.l[i]=new int[numNode];
    }
    fclose(fp);

    /* read length file*/
    if((fp=fopen(lengthFile,"r"))==NULL)
    {
        printf("can not open data file \%s\n",lengthFile);
        exit(-1);
    }
    fscanf(fp," %d",&temp);
    assert(numNode==temp);
    for(i=0;i<n.numNode;i++)
    {
        for(j=0;j<n.numNode;j++)
        {
            fscanf(fp," %d",&temp);
            n.p[i][j]=temp;
        }
    }
    for(i=0;i<n.numNode;i++)
    {
        for(j=0;j<n.numNode;j++)
        {
            fscanf(fp," %d",&temp);
            n.l[i][j]=temp;
            n.path[i][j]=new int[temp+1];
        }
    }
}
}
int *tp=new int[n.l[i][j]];
if(i==j) continue;
int num=getPath(n.p,i,j,tp,numNode);
//appending source
n.path[i][j][0]=i;
for(int k=0;k<num;k++) n.path[i][j][k+1]=tp[k];
delete tp;
}
return(n);
}

partition readPartition(int numNode,char* partFile, char* blockFile)
{
    partition ret;
    int numBlock;
    FILE *fp;

    ret.numNode=numNode;
    /* read block file in PAFN*/
    if((fp=fopen(blockFile,"r"))==NULL)
    {
        printf("can not open data file %s\n",blockFile);
        exit(-1);
    }
    fscanf(fp," %d",&numBlock);
    ret.numBlock=numBlock;
    ret.b=new int[numBlock];
    ret.n=new int[numBlock];
    for(int i=0;i<numBlock;i++)
    {
        fscanf(fp," %d",&ret.b[i]);
        ret.b[i]--; //b[i] starts from 1
        ret.n[i]=0;
    }
    fclose(fp);

    /* read partition result (block) file*/
    if((fp=fopen(partFile,"r"))==NULL)
    {
        printf("can not open partition file %s\n",partFile);
        exit(-1);
    }
    ret.r=new int[numNode];
    for(i=0;i<numNode;i++)
    {
        fscanf(fp," %d",&ret.r[i]);
        ret.n[ret.r[i]]++;
    }
    fclose(fp);
    return(ret);
}

/* evaluate the query cost of a shortest path over a block segmented broadcast sequence */
cost evaluatepath(int numBlock,int *bseq,int *bnum,int numNode,int *partition,int plen,int *path)
{
cost mycost;
int *cnt=new int[numBlock],nb=0;
for(int i=0;i<plen;i++)
{
    int bid=bseq[partition[path[i]]];
    for(int j=0;j<nb;j++)
        if(cnt[j]==bid) break;
    if(j==nb) cnt[nb++]=bid;
}
int maxind=-1,minind=numBlock;
//find the last block in the block sequence
int *seq=new int[numBlock];
for(i=0;i<nb;i++)
{
    for(int j=0;j<numBlock;j++)
        if(cnt[i]==bseq[j]) break;
    if(j==numBlock)
    {
        printf("error, no block found.......\n");
        exit(-1);
    }
    if(j>maxind) maxind=j;
    if(j<minind) minind=j;
}
mycost.t=nb;
mycost.a=0;
for(i=minind;i<=maxind;i++)
    mycost.a+=bnum[i];
/*printf("sp ");
for(i=0;i<plen;i++) printf("%3d",sp[i]);
printf("cnt ");
for(i=0;i<nb;i++) printf("%3d",cnt[i]);
printf("\n");*/
delete cnt;
delete seq;
return(mycost);
}
cost evaluateall(network n,partition p,int& count)
{
    assert(n.numNode==p.numNode);
    cost ret={0,0};
    count=0;
    for(int i=0;i<n.numNode;i++)
    for(int j=0;j<n.numNode;j++)
    {
        if(i==j) continue;
        cost stcost=evaluatepath
        {
            p.numBlock,
            p.b,
int uniqueNumber(int lower, int upper, int num, int *array)
{
    while(true)
    {
        int temp = lower + rand()%(upper-lower);
        for(int i=0;i<num;i++)
            if(array[i]==temp) break;
        if(i==num) return(temp);
    }
}

partition randomPartition(int numNode, int numBlock)
{
    partition p;
    p.numNode = numNode;
    p.numBlock = numBlock;
    p.r = new int [numNode];
    p.b = new int [numBlock];
    p.n = new int [numBlock];
    for(int i=0;i<numNode;i++)
        p.r[i] = rand()%numBlock;
    for(int i=0;i<numBlock;i++)
        p.b[i] = uniqueNumber(0,numBlock,i,p.b);
    for(int i=0;i<numNode;i++)
        p.n[i] = 0;
    for(i=0;i<numNode;i++)
        p.n[p.r[i]]++;
    return(p);
}

void main(int argc, char **argv)
{
    network n = readNetwork("ORGN.path","ORGN.length");
    //partition p = readPartition(n.numNode,"EAFN.graph.part.3","PAFN.block");
    FILE *fp1, *fp2;
    if((fp1=fopen("randtest.txt","w"))==NULL)
        exit(-1);
    if((fp2=fopen("equalgroup.txt","w"))==NULL)
        exit(-1);
    srand( (unsigned)time( NULL ) );
    for(int i=0;i<1000;i++)
{  
  partition p=randomPartition(n.numNode,3);  
  //printPS(p);  
  int count=0;  
  cost c=evaluateall(n,p,count);  
  fprintf(fp1,"%10d %10d",c.tt,c.at);  
  int equal=1;  
  for(int i=0;i<p.numBlock;i++)  
  {  
    fprintf(fp1,"%3d",p.n[i]);  
    if(p.n[i]!=4) equal=0;  
  }  
  fprintf(fp1,"n");  
  if(equal==1)  
  {  
    fprintf(fp2,"%10d %10d\n",c.tt,c.at);  
    freePS(p);  
  }  
  fclose(fp1);  
  fclose(fp2);  
  freeNS(n);  
  }