

Using PROC FCMP for Short Test Assembly

Tsung-hsun Tsai, Research League, LLC; Yung-chen Hsu, GED Testing Service

ABSTRACT

Assembling short test forms is a practical task in testing service organizations. A short test form contains less test items than the full-length test does, yet still preserves the essential test quality and captures the required test statistical characteristics. We explore an instance selection method for short test assembly task by using PROC FCMP to conduct low-level array operations to implement minimum spanning tree clustering. PROC FCMP was developed to allow SAS® users to write their own functions and subroutines for use in DATA step or SAS procedures. The purpose of this paper is to demonstrate how we can use SAS as a functional programming language by utilizing PROC FCMP to write reusable functions or subroutines to manage complex tasks.

INTRODUCTION

Constructing short test forms is a practical task in large or mid-sized testing service organizations. A short test form contains fewer test items than a full-length test form, yet still preserves the essential test quality and captures the required statistical characteristics. For example, a testing service organization often needs to publish test preparation materials, sample items, or study guides to help test takers prepare for the test. The sample test form or item samples may not have the same number of test items as the full-length form. There are other situations in the test development stage that the number of test items has to be reduced, too. A typical scenario encountered is that the testing time is not enough in the field testing, or the test is not able to fit the time restriction of the test session. In practice, there are also business factors, such as pricing strategy, cost consideration, and security issues. We explore a minimum spanning tree (MST) clustering approach to create short test forms by using PROC FCMP to conduct low-level array operations. SAS/OR® 12.1 and later versions do have a procedure to construct an MST easily. However, the flip side is it requires an extra cost for the license. SAS/IML®, a matrices language product, could be an ideal tool too, but again a license is required as well. Since the FCMP procedure already comes with SAS/BASE®, no additional cost is needed to achieve the goal. The purpose of this paper is to illustrate how we can leverage PROC FCMP to encapsulate the logic of algorithm in the reusable subroutines.

TEST CONSTRUCTION

Current test construction practices are mainly based on item response theory (IRT), a modern psychometric test theory that describes the relationship between item characteristics and test taker abilities. For example, the two-parameter logistic model (2PL) is one of the unidimensional IRT models for dichotomous responses used in various large-scale testing programs. The model can be expressed as

$$P_i(\theta, a_i, b_i) = \frac{1}{1 + \exp[-Da_i(\theta - b_i)]}$$

where P represents the probability of answering a particular dichotomously scored item correctly given the proficiency level θ , the parameters a_i and b_i are the characteristics of the item i , and D is the common choice of the scaling constant 1.7. Traditional approaches are based on item information function in constructing test forms. The item information function, which is derived from Fisher information, is defined as

$$I_i(\theta) = \frac{[P'_i(\theta)]^2}{P_i(\theta)(1 - P_i(\theta))}, \text{ where } P'_i(\theta) = \frac{\partial P_i(\theta)}{\partial \theta}$$

The test information of a test with n items is the sum of the item information:

$$I(\theta) = \sum_{i=1}^n I_i(\theta).$$

Mathematically, the task of test construction is to solve an optimization problem—to automatically find a suitable combination of items from item bank—a database—to assemble a test form. The traditional approaches to constructing short test forms are similar to the typical automated test assembly methods, which are based on the mathematical programming approach. Given the number of short test items, the test information is scaled down accordingly to form a new target information function. A short test form is then assembled in a similar fashion, but adopting the new target as the objective and the modified constraints. In practice, it has been well known that it is not always possible to arrive at a feasible solution through this approach. Therefore, turning the specifications or relaxing the constraints is usually unavoidable, and the effort is budgeted in the test development plan. Not only are the time

for turning the specifications often a concern in applying mathematical programming methods, computing costs and efficiency are also an issue. A more feasible strategy is an approach based on the Maximin Model (van der Linden & Boekkooi-Timminga, 1989). The idea is to use the shape of the target information function at certain ability points as the target without specifying a fix scaling factor. Said another way, the objective is to create a short test form to have a *similar test information curve*. This method has much higher chance to obtain a feasible solution because the formulation of the Maximum model is a special case of the goal programming approach (Hsu, 1993). Typically, the mean square deviation is used as a reference to measure the similarity between the short test forms and the original full-length form. Problem is, test forms with similar information curves do not warrant that the test items will have similar item characteristics. To circumvent the difficulty, Chen (2005) proposed a new view to treat the test assembly problem as a task of creating a joint distribution of item characteristics to have a *similar distribution* to the original distribution. Although the similarity is not explicitly defined in this view, it is conceptually appealing. Mathematically it provides a connection between a test form, as represented by the item characteristics, and geometrically, as represented by points in a multiple dimensional space. Similarly, with this view, the short test construction problem become an instance selection problem—to find a composition of points in a multiple dimensional space and this composition, albeit fewer points, represents the essential characteristics of the original distribution.

INSTANCE SELECTION

MST clustering method for instance selection is adopted here as an example to demonstrate the approach as MST is a reasonable method for clustering points in space into natural groups. There are other approaches, such as *k*-means clustering. However, the time complexity of *k*-means clustering is comparably more difficult and, unlike MST, the result depends on the initialization methods used in the algorithm.

In graph theory, a spanning tree of a connected, undirected graph is a sub-graph that contains all the vertices in such a way that there are no closed loops. Given a weight to each edge of the graph, an MST problem is to find a spanning tree of minimum total weight. The problem has many obvious applications and also arises in other guises, such as its close relation with single linkage cluster analysis (Gowe & Ross, 1969). MST is one of the best-studied optimization problems in computer science. The idea of using MST algorithm to cluster points into groups is outlined as follows. First, an MST is constructed. If the longest edge is removed, the original tree becomes two sub-trees. The longest edge of the trees in the forest is removed repeatedly in the forest until the point the number of trees in equals the desired number. In the end, each sub-tree is a cluster of points. For short test form assembly, each cluster is replaced by a point to represent the cluster, such as the centroid of the points or a random selection among the points in the cluster. Finally, a virtual test form is obtained, which contains the desired number of items, to serve as a template for querying test items from an item bank to assemble test forms.

MST CLUSTERING ALGORITHMS IN FCMP SUBROUTINES

There are two classic algorithms for finding an MST: Kruskal's (1956) algorithm and Prim's (1957) algorithm. In general, the number of vertices matter more in Prim's algorithm while, on the other hand, the Kruskal' algorithm works best if the number of edges can be kept to a minimum in the adjacency matrix. SAS/OR® 12.1 and later versions use Kruskal's algorithm to construct MST. However, for short test form assembly, the adjacency matrix is a dense matrix, so Prim's algorithm is relatively a better choice. In fact, the implementation is often assigned as a programming exercise in computer science courses due to its simplicity. It is possible to use SAS/IML® to implement the MST algorithm. However, PROC FCMP functions and subroutines are more feasible than SAS/IML® for they can be easily encapsulated in DATA steps. The algorithm is implemented as a PROC FCMP subroutine `MST` for two dimension graphs.

```
*** Construct MST;
subroutine MST(nameInf $,nameOutf $);
  array wt[1,1] /nosymbols;
  rc=read_array(nameInf,wt);
  nNode=dim(wt);
  wtMax=wt[1,1];
  do i=1 to nNode;
    do j=1 to nNode;
      if wt[i,j] ~=. and wt[i,j] > wtMax then wtMax=wt[i,j];
    end;
  end;
  wtMax=ceil(wtMax);          *upper limit;

  * Init;
  array tr[1]      /nosymbols;          *tree;
  array idx[1]     /nosymbols;          *MST index;
  array iNode[1]   /nosymbols;          *node status;
  array wtTmp[1]   /nosymbols;          *temporary weight;
```

```

call dynamic_array(tr,nNode-1,3);
call dynamic_array(idx,nNode);
call dynamic_array(iNode,nNode);
call dynamic_array(wtTmp,nNode);
do i=1 to nNode;
    wtTmp[i]=wtMax;
    iNode[i]=0;
end;
idx[1]=0;                                *init first node;
wtTmp[1]=0;                              *init temp weight;

* Find minimum weights;
do i=1 to nNode;
    wtMin=wtMax;
    do j=1 to nNode;
        if iNode[j]=0 and wtTmp[j]<=wtMin then do;
            wtMin=wtTmp[j];                *update minimum weight;
            ii=j;
        end;
    end;
    iNode[ii]=1;                          *update node status;

* Update MST indices and weights;
do j=1 to nNode;
    if (iNode[j]=0 and wt[ii,j]~=0 and wt[ii,j]<wtTmp[j]) then do;
        idx[j]=ii;
        wtTmp[j]=wt[ii,j];
    end;
end;

end;

* Store MST;
do i=2 to nNode;
    tr[i-1,1]=idx[i];
    tr[i-1,2]=i;
    tr[i-1,3]=wt[i,idx[i]];
end;
rc=write_array(nameOutf,tr,'node1','node2','weight');
endsub;

```

The subroutine `MST` takes the adjacency matrix as the first argument and outputs the result in the dataset `nameOutf`. The dataset `nameOutf` contains the links that belong to an MST. Notable in this subroutine are:

1. The arrays are specified as `nosymbols`, which means we only use array subscripting to access the elements, because we do not know the size of the arrays in advance,
2. The `read_array` function will dynamically resize the array to fit the dimensions of the input dataset after the adjacency matrix is read.
3. After knowing the size of adjacency matrix, the following four arrays are resized to the provided dimensions by using the `dynamic_array` function.

In the following code, a PROC FCMP subroutine `CLUS` links edges of the connected nodes together to form clusters (see Appendix 2). This example creates a macro called `MST_CLUS` to perform clustering task. At the beginning, the MST dataset is sorted by weight. To remove the longest edges, the dataset is split into two sets, a set of isolated nodes without an edge, and the second set pairs nodes with an edge. The second set is further processed with subroutine `CLUS`. It is possible to execute the subroutine from PROC FCMP to perform the task directly. The macro demonstrates the feasibility to call the subroutine within a DATA step.

```

/* MST clustering */
%macro MST_CLUS(nClus=,mMst=,iNode=,mClus=);
proc sort data=&mMst out=_matMst;
    by descending weight;
run;
data _matMst1 _matMst0; set _matMst(drop=weight);

```

```

        if _n_ < &nClus then output _matMst0;
        else output _matMst1;
run;

* Isolated nodes;
data _tmp1(keep=idNode);
    set _matMst1;
    idNode=node1; output;
    idNode=node2; output;
run;
data _tmp0(keep=idNode);
    set _matMst0;
    idNode=node1; output;
    idNode=node2; output;
run;
proc sort data=_tmp1 nodup; by idNode; run;
proc sort data=_tmp0 nodup; by idNode; run;
data &iNode(rename=(idNode=cluster));
    merge _tmp1 (in=in1) _tmp0 (in=in0);
    by idNode;
    if (in1=0 and in0=1) then output;
run;

* Clustered nodes;
proc sort data=_matMst1; by node1 node2; run;
data _null_;
    call CLUS("_matMst1", "&mClus");
run;

* Clean up;
proc datasets lib=work nolist;
    delete _tmp0 _tmp1 _matMst _matMst0 _matMst1;
quit;
run;
%mend MST_CLUS;

```

A SIMULATION EXAMPLE

To demonstrate an implementation, we simulated a SAS dataset `refts` that contains 100 pairs of parameters, which represent a 100-item full-length test form based on the two-parameter logistic model (2PL). Our target is to create a 50-item virtual test form. The similarity measure for a pair of items i and j is defined as

$$d_{ij} = \sqrt{(b_i - b_j)^2 + (\log(a_i) - \log(a_j))^2},$$

which is essentially the Euclidean distance. We purposely used 2PL in this example for displaying MST in a two dimensional plan, b versus $\log(a)$, to better illustrate the clustering idea. In the following code, the `ADJ_MAT` subroutine uses the dataset `refts` to generate the adjacency matrix `matAdj` (see Appendix 1). After the MST construction and clustering steps, the `CENTROID` subroutine converts the points in a cluster into a single point by computing the centroid of each cluster (see Appendix 1).

```

libname sesug 'C:\sesug\dat';
data refts; set sesug.refts; run;

options cmplib=work.funcs;
data _null_;
    *** Compute adjacency matrix;
    call ADJ_MAT("refts", "matAdj");
    *** Generate MST;
    call MST("matAdj", "matMst");
run;

*** Separate isolated and clustered nodes;
%MST_CLUS(nClus=50, mMst=matMst, iNode=isoNode, mClus=clusNode);

*** Compute centroids;

```

```

* Centroid coordinates;
data _null_;
  call CENTROID("clusNode","refts","corrCtrd");
run;
* Point coordinates;
data _tmp1; set refts; cluster=_n_; run;
data corrIso(drop=cluster);
  merge _tmp1 (in=in1) isonode (in=in0);
  by cluster;
  if (in1=1 and in0=1) then output;
run;
* Combine isolated points and centroids;
data corr; set corrCtrd corrIso; run;

```

Figure 1 illustrates the scatter plot of $\log(a)$ versus b that represents the original 100-item full-length test form. Although edges are not shown in the graph, an MST is constructed based on the assumed graph that each point is connected with the other points as an undirected graph.

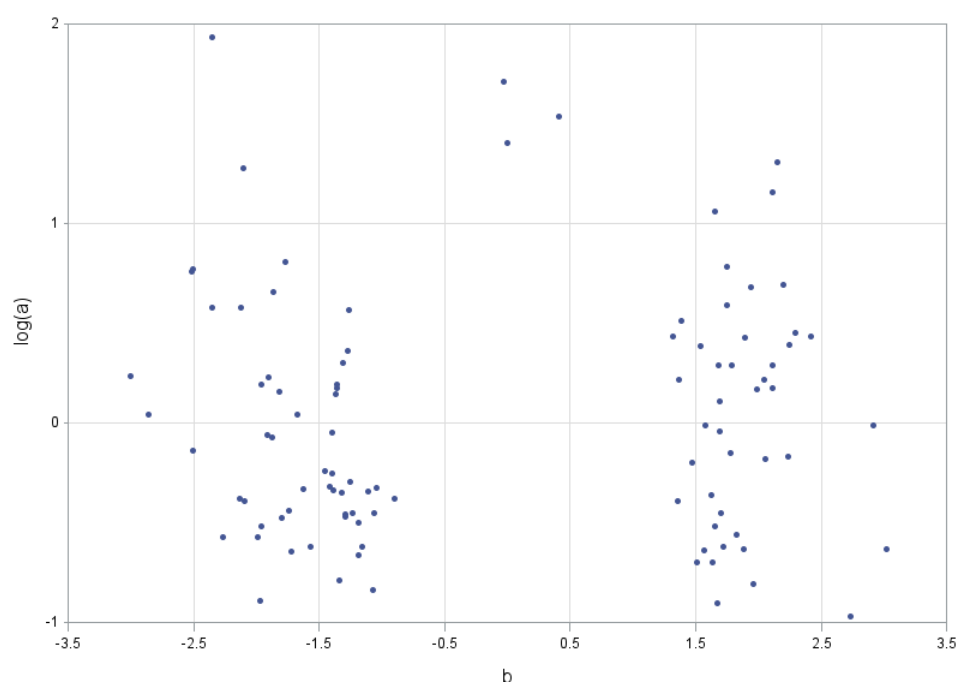


Figure 1. Scatter of $\log(a)$ versus b of the original full-length test form.

Figure 2 is the MST and figure 3 illustrates the clusters after removing the 49 longest edges from the initial MST. We end up with 50 clusters.

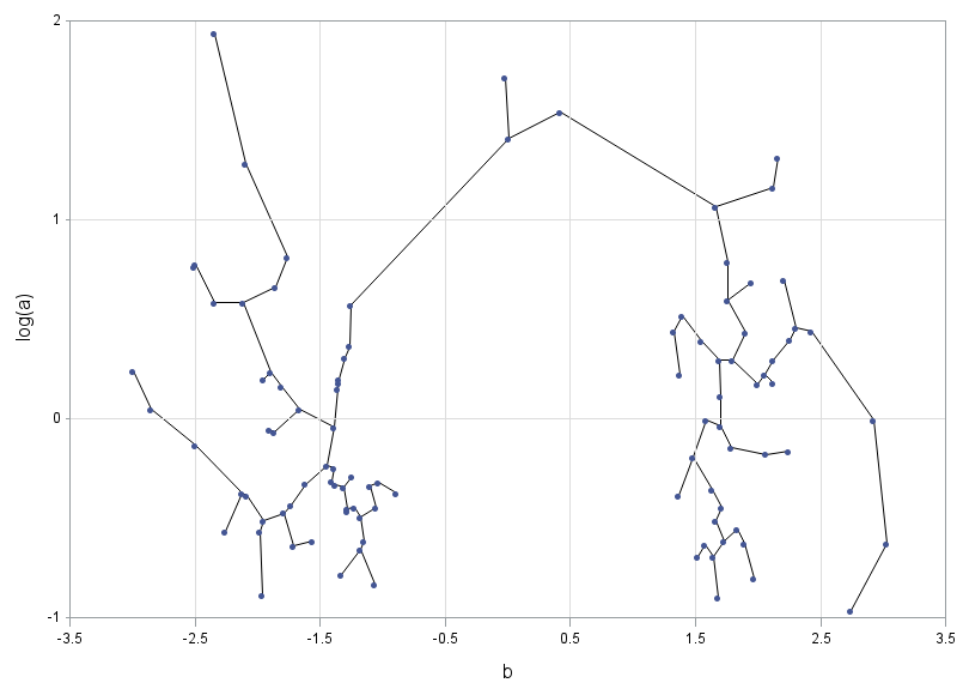


Figure 2. MST of figure 1

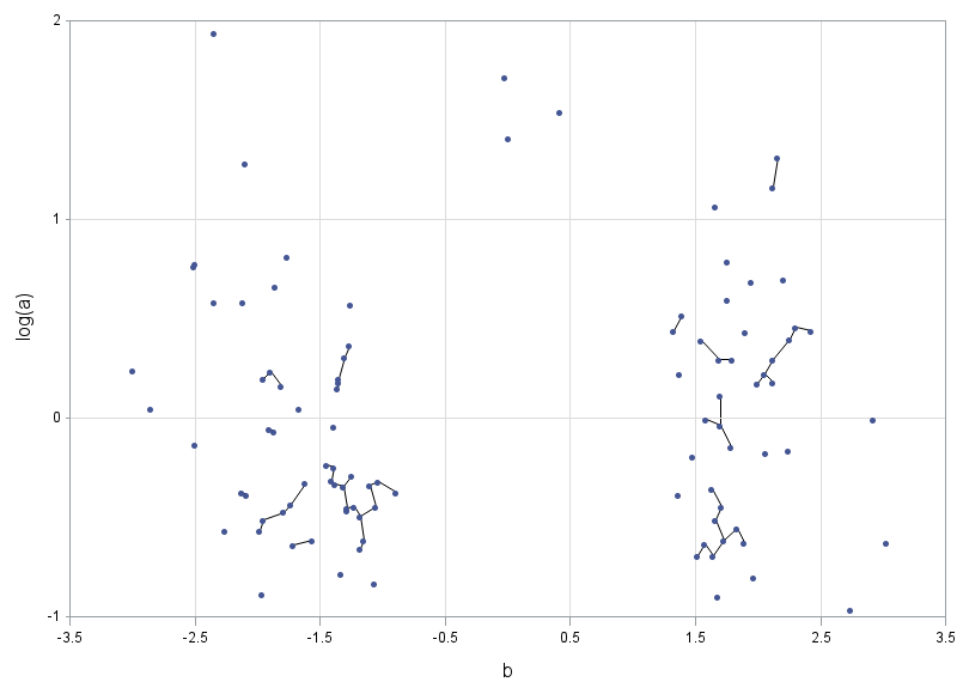


Figure 3. Illustration of the 50 clusters after removing the 49 longest edges from Figure 2.

By replacing the clusters with their centroids, a 50-item template is obtained as shown in figure 4 (see Appendix 3 for details about generating the figures).

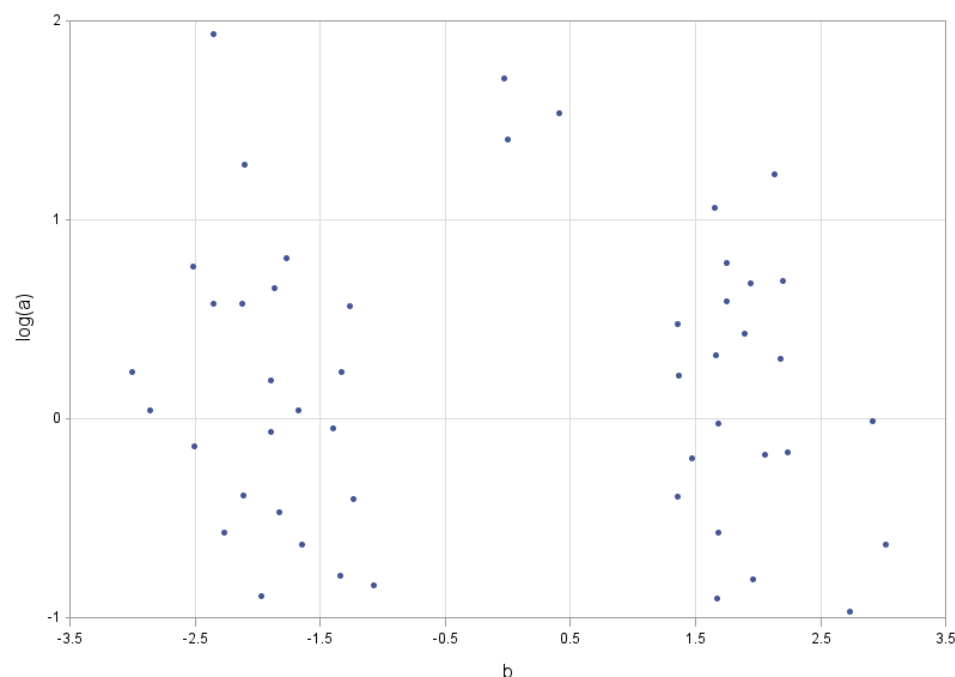


Figure 4. Virtual test form as the points in the same cluster is replaced by their centroid from Figure 3.

CONCLUSION

In this paper, we use short test form assembly problem as an example to show the flexibility of FCMP procedure as a development tool. PROC FCMP is powerful and is part of SAS/BASE®, which means it is available with any SAS system without the need to purchase an extra license. FCMP procedure enables SAS programmers to wrap computations as reusable and interoperable functions or subroutines. And what's more, PROC FCMP encourages modular approach to allow programmers to develop their own code independently, reuse the code in subsequent code, and share the code between programs and other users, which greatly eases code management for complex project. Comparing to DATA step array, PROC FCMP supports both row-wise and column-wise index notation for complex low-level array operations.

REFERENCES

- Chen, P.-H. (2005). *IRT-based automated test assembly: A sampling and stratification perspective* (Doctoral Dissertation). University of Texas, Austin, TX
- Gower, J. C., Ross, G. J. S. (1969). Minimum spanning trees and single linkage cluster analysis. *J. Roy. Statist. Soc. Ser. C*, 18 (1), 54-64.
- Hsu, Y.-C. (1993). *The goal programming approach for test construction* (Master thesis). The University of Arizona, Tucson, AZ.
- Kruskal, J. B. Jr. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.* 7, 48-50.
- van der Linden, W. J., & Boekkooi-Timminga, E. (1989). A maximin model for test design with practical constraints. *Psychometrika*. 54, 237-247.
- Prim, R. C. (1957). Shortest connection networks and some generalization. *Bell Sys. Tech. J.* 36, 1389-1401.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Tsung-hsun Tsai
 Research League, LLC
 11 Crimson King Drive, Holmdel, NJ 07733
 E-mail: ttsai@researchleague.org

Yung-chen Hsu
 GED Testing Service
 1919 M Street, NW – Suite 600, Washington, DC 20036
 E-mail: yung-chen.hsu@GEDtestingservice.com
 Web: www.gedtestingservice.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX 1

```
proc fcmp outlib=work.funcs.tc;
  *** Compute adjacency matrix;
  subroutine ADJ_MAT(nameInf $, nameOutf $);
    array pt[1,1] /nosymbols;
    array x[1] /nosymbols;
    rc=read_array(nameInf,pt);
    nr=dim(pt);
    call dynamic_array(x,nr,nr);
    do i=1 to nr-1;
      do j=i+1 to nr;
        da=pt[i,1]-pt[j,1];
        db=pt[i,2]-pt[j,2];
        x[i,j]=da*da+db*db;
        x[j,i]=x[i,j];
      end;
    end;
    rc=write_array(nameOutf,x);
  endsub;

  *** Compute Centroid;
  subroutine CENTROID(dsPt $, dsCorr$, dsCtrd $);
    array id[1,1] /nosymbols;          *point id;
    rc=read_array(dsPt,id);
    nr=dim1(id);
    nc=dim2(id);
    array corr[1] /nosymbols;          *point coordinate;
    rc=read_array(dsCorr,corr);
    array ctrd[1,2] /nosymbols;        *centroid coordinate;
    call dynamic_array(ctrd,nr,2);

    do i=1 to nr;
      ctrd[i,1]=0;
      ctrd[i,2]=0;
    end;
    do i=1 to nr;
      cntPt=0;
      do j=1 to nc;
        if id[i,j]~=. then do;
          ctrd[i,1]=ctrd[i,1]+corr[id[i,j],1];
          ctrd[i,2]=ctrd[i,2]+corr[id[i,j],2];
          cntPt=cntPt+1;
          if j=nc then do;
            ctrd[i,1]=ctrd[i,1]/(cntPt*1.0);
            ctrd[i,2]=ctrd[i,2]/(cntPt*1.0);
          end;
        end;
      end;
      else do;
        ctrd[i,1]=ctrd[i,1]/(cntPt*1.0);
        ctrd[i,2]=ctrd[i,2]/(cntPt*1.0);
      end;
      j=nc;
    end;
  end;
```

```

    end;
end;
rc=write_array(dsCtrd,ctrd,'a','b');
endsub;
quit;

```

APPENDIX 2

```

proc fcmp outlib=work.funcs.mst;
*** Perform MST clustering;
subroutine CLUS(nameInf $,nameOutf $);
  * Read links;
  array M[1,1] /nosymbols;
  rc=read_array(nameInf,M);

  * Init;
  nClus=dim(M);
  array C[1] /nosymbols;          *cluster matrix;
  array szClus[1] /nosymbols;     *cluster size;
  rPtr=1;                         *row pointer;
  call dynamic_array(C,nClus-1,nClus);
  call dynamic_array(szClus,nClus);
  do i=1 to nClus;
    szClus[i]=0;
  end;
  C[1,1]=M[1,1];
  C[1,2]=M[1,2];
  szClus[1]=2;

  * Loop through each link;
  do k=2 to nClus;
    m1=0;                         *init match flags;
    m2=0;
    * Match 1st node;
    do i=1 to rPtr;
      do j=1 to szClus[i];
        if M[k,1]=C[i,j] then do;
          m1=1;
          x1=i;                   *mark row number;
          n1=M[k,2];             *store 2nd node;
          j=szClus[i];
          i=rPtr;
        end;
      end;
    end;
    * Match 2nd node;
    do i=1 to rPtr;
      do j=1 to szClus[i];
        if M[k,2]=C[i,j] then do;
          m2=1;
          x2=i;
          n2=M[k,1];
          j=szClus[i];
          i=rPtr;
        end;
      end;
    end;
    * Link nodes or clusters;
    if m1=0 and m2=0 then do;
      rPtr=rPtr+1;
      C[rPtr,szClus[rPtr]+1]=M[k,1]; *add a cluster;
      C[rPtr,szClus[rPtr]+2]=M[k,2];
      szClus[rPtr]=szClus[rPtr]+2;
    end;
    else if m1=0 then do;

```

```

        szClus[x2]=szClus[x2]+1;
        C[x2,szClus[x2]]=n2;           *link node to cluster;
    end;
    else if m2=0 then do;
        szClus[x1]=szClus[x1]+1;
        C[x1,szClus[x1]]=n1;           *link node to cluster;
    end;
    else do;
        do i=1 to szClus[x2];
            C[x1,szClus[x1]+i]=C[x2,i]; *link clusters;
            C[x2,i]=.;
        end;
        szClus[x1]=szClus[x1]+szClus[x2];
        szClus[x2]=0;
    end;
end;

* Output Clusters;
maxSzClus=szClus[1];                  *maximum size of cluster;
if szClus[1]~=0 then nSzClus=1;
else nSzClus=0;
do i=2 to nClus-1;
    if szClus[i]> maxSzClus then maxSzClus=szClus[i];
    if szClus[i]~=0 then nSzClus=nSzClus+1;
end;
array cluster[1] /nosymbols;
call dynamic_array(cluster,nSzClus,maxSzClus);
k=1;
do i=1 to nClus-1;
    if szClus[i]~=0 then do;
        do j=1 to szClus[i];
            cluster[k,j]=C[i,j];
        end;
        k=k+1;
    end;
end;
rc=write_array(nameOutf,cluster);
endsub;
quit;

```

APPENDIX 3

```

* Annotate data sets;
%let nClus=50;
proc sort data=matMst out=_ds0;
    by descending weight;
run;
data _ds0 (drop=weight);
    set _ds0;
    if _n_<&nClus then link=0; else link=1;
run;
data _ds1a; set refts(rename=(a=a1 b=b1)); n1=_n_; run;
data _ds1b; set refts(rename=(a=a2 b=b2)); n2=_n_; run;
proc sql;
    create table _anno0 as
    select * from _ds0, _ds1a where node1=n1;
quit;
proc sql;
    create table _anno1 as
    select * from _anno0, _ds1b where node2=n2;
quit;
data _anno2; set _anno1; if link=0 then delete; run;

data mst1; set _anno1;
    length function style color $8;

```

```

    xsys='2'; ysys='2';
    function='move'; x=b1; y=a1; output;
    function='draw'; color=black; line=1; size=1; x=b2; y=a2; output;
run;
data mst2; set _anno2;
    length function style color $8;
    xsys='2'; ysys='2';
    function='move'; x=b1; y=a1; output;
    function='draw'; color=black; line=1; size=1; x=b2; y=a2; output;
run;
proc datasets library=work nolist; delete _; quit;

axis1 label=(angle=90 height=1.3 'log(a)') order=(-1 to 2 by 1) minor=none
offset=(0,0);
axis2 label=(height=1.3 'b') order=(-3.5 to 3.5 by 1) minor=none offset=(0,0);
symbol1 font='Symbol MT' value='b7'x height=2.3pct interpol=none;

* Scatter plot (figure 1);
proc gplot data=refts;
    plot a*b=1 /
        vaxis=axis1 haxis=axis2
        autoref chref=graydd autovref cvref=graydd;
run;
* MST plot (figure 2);
proc gplot data=refts;
    plot a*b=1 /
        vaxis=axis1 haxis=axis2
        autoref chref=graydd autovref cvref=graydd
        annotate=mst1;
run;
* Cluster plot (figure 3);
proc gplot data=refts;
    plot a*b=1 /
        vaxis=axis1 haxis=axis2
        autoref chref=graydd autovref cvref=graydd
        annotate=mst2;
run;
* Centroid plot (figure 4);
proc gplot data=corr;
    plot a*b=1 /
        vaxis=axis1 haxis=axis2
        autoref chref=graydd autovref cvref=graydd;
run;
quit;

```