

Computing Risk Measures for Cross Collateralized Loans Using Graph Algorithms

Chaoxian Cai, Automated Financial Systems, Inc.

ABSTRACT

In commercial lending, multiple loans or commitments may be collateralized by multiple properties, and this cross collateralization creates linked loan-property networks. These networks are embedded in a loan-property table, stored in a relational database. Some risk measures for these cross collateralized loans or commitments are better evaluated in aggregated terms if we can identify all cross linked properties, loans, and commitments. The Union-Find algorithm is commonly used to find connected components in a graph. In this paper, I have implemented the primary operations of the Union-Find algorithm in a SAS® macro program using only Base SAS DATA steps. The program can be used to find all connected components in a SAS data set and separate them into discrete groups. It can be applied to find all cross collateralized loan-property networks and compute pooled loan-to-value (LTV) ratios. The program can also be applied to identify main obligations and loan structures of future commitments with takedown loans and hierarchical lines of credit. Risk measures, such as exposure at default (EAD) and credit conversion factor (CCF), are computed for these complicated loans and illustrated by examples.

INTRODUCTION

A graph $G(V, E)$ is a collection of vertices and edges that defines the links between vertices. A graph is connected if every pair of vertices is connected by a path. A connected component is a subgraph that has every vertex reachable from other vertices. A complex network is naturally represented by a graph, which, in graph terminology, uses vertices to represent individual objects and edges to represent the relationships between any two distinct objects. Vertices and edges in a graph are also commonly referred as nodes and paths, respectively. In a relational database, a graph is implicitly represented by columns and rows that bearing relationships (edges) between two distinct columns (vertices). A SAS data set is a relational table with variables as columns and observations as rows. To store a complex network in a relational table, we need to create a column attribute for each object of interest and add a row observation for each link between two objects. In this way, complex networks can be conveniently stored in a relation table such as a SAS data set.

In this paper, I will present a SAS macro program that can be used to identify all connected components from a SAS data set and group all connected components into discrete groups. I will show you how to apply the macro program to solve a few problems in the banking industry. These examples include finding cross collateralized loans in a loan portfolio, computing pooled loan-to-value ratios, finding main obligations of future commitments, and computing exposures at default and credit conversion factors.

EXAMPLE

Let us start with a simple example to illustrate the problem. The graph in Figure 1 has 18 vertices and 13 edges, and the 5 connected components are {1, 2, 3, 11, 12}, {4, 5, 6, 13, 14}, {7, 8, 15}, {9, 16, 17}, and {10, 18}, respectively.

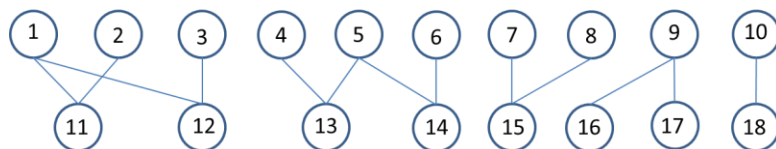


Figure 1. A graph with 5 connected components.

The graph in Figure 1 can be easily read into a SAS data set with the following code:

```
data linkednodelist;
  input node_x node_y @@;
  datalines;
1 11 1 12 2 11 3 12 4 13 5 13 5 14 6 14 7 15 8 15 9 16 9 17 10 18
;
```

The resulted data set (*linkednodelist*) is showed in Table 1, where every link between two distinct vertices has a row entry in the table. The graph algorithm to find connected components will use this table as the input. In business applications, such a SAS data set will often consist of thousands to millions of nodes and links, and the graphical structures in such a big data set are hidden and unknown to the users. Discovering hidden graphical structures in a big data set certainly is of interest to the data miners; however, finding all graphical structures in a large SAS data set

with millions of rows is computationally demanding and challenging.

<i>Obs</i>	<i>node_x</i>	<i>node_y</i>
1	1	11
2	1	12
3	2	11
4	3	12
5	4	13
6	5	13
7	5	14
8	6	14
9	7	15
10	8	15
11	9	16
12	9	17
13	10	18

Table 1. Data set *linkednodelist*.

THE MACRO PROGRAM

The Union-Find algorithm is a classic graph algorithm used to join disjoint sets and find connected components of a graph from a given set of vertices and edges (Weiss, 1994; Cormen, 1997). The algorithm requires tree data structures to store connected components. Since SAS does not have tree-type data structures, the implementation of Union-Find algorithm requires an implicit set up of tree structures using one-dimensional array. Previously, I had presented a paper on the title of “Implementing Union-Find Algorithm with Base SAS DATA Steps and Macro Functions”, where I gave a detailed explanation of the algorithm, how the algorithm works, and the programming techniques used to implement the algorithm (Cai, 2015). In this paper, I have wrapped the implementation of Union-Find algorithm in a macro function named `%group_connected_components`. The complete macro program is listed in APPENDIX A. Following are the macro parameters and major steps inside the macro function:

```
%macro group_connected_components(edge_list=, vx=, vy=, elistout=elistout,
vlistout=vlistout, hierarchy=FALSE);

/* Step 1: a) Create vertex list from the input edge list;
           b) Label each vertex sequentially from 1 to N;
           c) Create new edge list with each vertex labeled.
*/

/* Step 2: a) Process edge list with Union-Find algorithm;
           b) Output connected components.
*/

/* Step 3: a) Output updated vertex list with connected components IDs;
           b) Output updated edge list with connected components IDs;
*/

%mend group_connected_components;
```

The input data is an edge list data set (`&edgelist`) which has vertices defined by two columns (`&vx` and `&vy`); the data set `&elistout` is the output edge list with unique *ConcompID* assigned for each connected component, and the data set `&vlistout` is a vertex list with each vertex (object) identified by *ObjectID* and has row entries of *NodeID* and *ConcompID*. The data set can be set up as a hash table to find the connect component *ConcompID* for each vertex *NodeID*. The macro function includes the following steps: 1) create new vertex list (`vlist`) and edge list (`elist`) and label each vertex with a unique sequential number (*NodeID*), 2) process the edge list (`elist`) and apply the Union-Find algorithm to build connect components, and 3) output the resulted vertex list (`&vlistout`) and edge list (`&elistout`). There are three basic set operations in the Union-Find algorithm: making disjoint sets, finding disjoint sets, and merging disjoint sets. Initially, each vertex is treated as a disjoint set, and N disjoint sets are set up at the beginning for N numbers of vertices. In each DATA step iteration, an edge that links two vertices (`&vx` and `&vy`) is read, and the disjoint sets that contain the vertices `&vx` and `&vy` are found and merged. The DATA step loop continues until all edges have been processed.

The Union-Find algorithm is for undirected graph. It employs union-by-depth optimization to improve the run time efficiency. If the connected components have hierarchical structures, the union-by-depth operations will break the hierarchical relation between a child node and its parent node. If there are needs to preserve the hierarchical structures, we need to join two disjoint sets based on an edge's hierarchical direction. In this case, we will always let

the child node pointing to the parent node when two subtrees are merged together. In the macro function, `hierarchy=FALSE` is set as default, which will use the union-by-depth operations. Setting `hierarchy=TRUE` will use the union operations without union-by-depth optimization and preserve tree hierarchies of all connected components.

The macro function can be conveniently called to find connected components from a SAS data set. For example, to find all connected components in Table 1, you can call the macro function as follows:

```
%group_connected_components(edgelist=linkednodelist, vx=node_x, vy=node_y,
elistout=result)
```

The result is shown in Table 2. The column *ConcompID* is added to identify each connected component. Note that each distinct connected component has been assigned a unique *ConcompID*. A connected component is identified by the root node of the tree. Note that the 5 connected components: {1, 2, 3, 11, 12}, {4, 5, 6, 13, 14}, {7, 8, 15}, {9, 16, 17}, and {10, 18}, are correctly identified by *ConcompID* 2, 5, 8, 9, and 10, respectively.

<i>Obs</i>	<i>node_x</i>	<i>node_y</i>	<i>ConcompID</i>
1	1	11	2
2	1	12	2
3	2	11	2
4	3	12	2
5	4	13	5
6	5	13	5
7	5	14	5
8	6	14	5
9	7	15	8
10	8	15	8
11	9	16	9
12	9	17	9
13	10	18	10

Table 2. The output data set after calling `%group_connected_components`.

THE PROC OPTNET PROCEDURE

Finding connected components in a graph can also be solved by using SAS/OR® software. PROC OPTNET is one of procedures in SAS/OR software that has implemented a number of graph algorithms for combinatorial optimizations and network analyses. Given a graph with a set of edges and vertices, PROC OPTNET can be used to solve graph problems such as finding connected components, detecting cycles in a graph, discovering shortest path, and so on. Here we will discuss how to use PROC OPTNET to find connected components.

To find connected components in a data set, *concomp* statement is used in PROC OPTNET procedure. The *data_links=* option specifies the link data set *linkednodelist*, the *data_link_var* options *from=* and *to=* specify the data set variable name for the *from* nodes and the data set variable name for the *to* nodes, respectively.

```
proc optnet
  data_links = linkednodelist
  out_nodes = result2;
  data_links_var from = node_x
                  to   = node_y;
  concomp;
run;
```

The output data set *result2* is a node list table with 2 named columns: *node* and *concomp*. The listing output is shown in Table 3. There are 5 distinct connected components that have been labeled sequentially from 1 to 5. In comparison to the result in Table 2, the column *node* corresponds to the columns *node_x* and *node_y*, the column *concomp* corresponds to the column *ConcompID*, and each connected component consists of exactly the same set of nodes.

<i>Obs</i>	<i>node</i>	<i>concomp</i>
1	1	1
2	11	1
3	12	1
4	2	1
5	3	1
6	4	2
7	13	2
8	5	2

9	14	2
10	6	2
11	7	3
12	15	3
13	8	3
14	9	4
15	16	4
16	17	4
17	10	5
18	18	5

Table 3. The output data set from PROC OPTNET run.

APPLICATIONS IN COMPUTING RISK MEASURES

COMPUTING LOAN-TO-VALUES FOR CROSS COLLATERALIZED LOANS

The term of loan-to-value (LTV) is a key risk measure in real estate lending and is defined as the ratio of the amount of a loan to the current value of the collateralized property. Banks often set up LTV guidelines when issuing new loans to limit risk exposures. For example, for commercial loans, the supervisory LTV upper limit is often set to be 80%, which means that a bank will not issue a loan if the loan amount is above 80% of the pledged collateral (OCC, 2013).

A single loan may be securitized by a single or multiple properties, and a highly valued property may be used to secure multiple loans. If a single or multiple properties are used to secure only a single loan, the calculation of LTV is simple and can be obtained by dividing the amount of loan by the total collateralized property value. On the other hand, if a single or multiple properties are used to secure multiple loans, these multiple loans may have different levels of seniority or priority and the calculation of LTV is complicated (Ono et al., 2013). To illustrate the calculation, in Table 4, I have listed some conceived examples of loans and collateralized properties that are currently in a bank's loan portfolio. Loan L01 is collateralized by a single property P01, and L02 is collateralized by 2 properties, P05 and P06. In these two cases, the LTV ratio is calculated by dividing the loan amount by the total property values. Loan L03 is collateralized by property P04, and was originated on March 1, 2010, and loan L04 is collateralized by the same property P04, and was originated on a late date, April 1, 2015. If we consider loan seniority and assume that a senior loan has higher priority over a junior loan, then the LTVs of L03 and L04 are $N3/V4$ and $(N3+N4)/V4$, respectively. If we do not consider loan priority, then for both loans, the current pooled LTV ratio is $(N3+N4)/V4$.

Obs	Loan ID	Loan Amount	Origination Date	Property ID	Property Value	LTV (prioritized)	LTV (pooled)
1	L01	N1	01May2010	P01	V1	$N1/V1$	$N1/V1$
2	L02	N2	01Feb2010	P05	V5	$N2/(V5+V6)$	$N2/(V5+V6)$
3	L02	N2	01Feb2010	P06	V6	$N2/(V5+V6)$	$N2/(V5+V6)$
4	L03	N3	01Mar2010	P04	V4	$N3/V4$	$(N3+N4)/V4$
5	L04	N4	01Apr2015	P04	V4	$(N3+N4)/V4$	$(N3+N4)/V4$
6	L05	N5	01Jun2010	P02	V2	$N5/V2$	$(N5+N6+N7)/(V2+V3)$
7	L06	N6	01Sep2012	P02	V2	Not defined	$(N5+N6+N7)/(V2+V3)$
8	L06	N6	01Sep2012	P03	V3	Not defined	$(N5+N6+N7)/(V2+V3)$
9	L07	N7	01Oct2015	P03	V3	$(N6+N7)/V3$	$(N5+N6+N7)/(V2+V3)$

Table 4. Some examples of loans and collateralized properties in a bank's loan portfolio.

In Table 4, loans L05 and L06 are both collateralized by property P02, and loan L05 was underwritten prior to loan L06; similarly, loans L06 and L07 are both collateralized by property P03, and loan L06 was underwritten prior to loan L07. Here, loan L06 is collateralized by 2 properties, P02 and P03, and L06 is a junior loan regarding to L05 but a senior loan regarding to L07. In this case, we cannot define the LTV ratio for loan L06 if we consider loan priority. However, if we do not consider loan priority, the calculation is less complicated, and the pooled LTV is $(N5+N6+N7)/(V2+V3)$ for all loans that are cross collateralized.

A bank's loan portfolio often contains several thousands to several millions of accounts, and loan-to-property relationships of one-to-one, one-to-many, many-to-one, and many-to-many are common scenarios. The one-to-one loan-to-property relationship is defined in a loan property table by a row entry with loan and property as two column attributes; the one-to-many, many-to-one, and many-to-many loan-to-property relationships are embedded in a loan

property relational table with multiple row entries. To compute the LTV ratio for each loan in a relation table, all loan-property pairs and their connected components need to be discovered and identified first. The following code creates a SAS data set named *loanprop* with some conceived numbers for loan amounts and property values. The macro function *%group_connected_components* is invoked to find all connected components.

```
data loanprop;
    input loan_id $ property_id $ loan_amount property_value;
    datalines;
L01 P01 1.5 2.0
L02 P05 1.0 1.0
L02 P06 1.0 0.5
L03 P04 0.5 3.0
L04 P04 0.25 3.0
L05 P02 0.3 1.0
L06 P02 1.0 1.0
L06 P03 1.0 5.0
L07 P03 0.75 5.0
;

%group_connected_components(edgelist=loanprop, vx=loan_id, vy=property_id,
elistout=loanprop_pool)
```

After identifying all connected components in Table 4, the computation of LTVs is straightforward. We just add the loan amounts from all distinct loans within each connected component to get the group loan amount. We do the same aggregation for the properties to get the group property value. For each group, the LTV ratio is computed by dividing the group loan amount by the group property value. The following SAS code is used to calculate the pooled LTV ratios for cross-collateralized loans.

```
proc sql noprint;
create table loans as
select distinct ConcompID,
               loan_id,
               loan_amount
from loanprop_pool;

create table loan_sums as
select distinct ConcompID,
               sum(loan_amount) as g_loan_amount
from loans
group by ConcompID;

create table properties as
select distinct ConcompID,
               property_id,
               property_value
from loanprop_pool;

create table property_sums as
select distinct ConcompID,
               sum(property_value) as g_property_value
from properties
group by ConcompID;

create table loan_ltv as
select a.*,
       b.g_loan_amount,
       c.g_property_value,
       b.g_loan_amount/c.g_property_value as LTV format=3.2
from loanprop_pool as a
left join loan_sums as b
on a.ConcompID=b.ConcompID
left join property_sums as c
on a.ConcompID=c.ConcompID
order by loan_id;
quit;
```

The results are shown in Table 5. The algorithm has identified 4 connected components in data set *loanprop*. The pooled LTV is computed for each component.

Obs	ConcompID	loan_id	property_id	loan_amount	property_value	g_loan_amount	g_property_value	LTV
1	1	L01	P01	1.50	2.0	1.50	2.0	.75
2	2	L02	P05	1.00	1.0	1.00	1.5	.67
3	2	L02	P06	1.00	0.5	1.00	1.5	.67
4	4	L03	P04	0.50	3.0	0.75	3.0	.25
5	4	L04	P04	0.25	3.0	0.75	3.0	.25
6	5	L05	P02	0.30	1.0	2.05	6.0	.34
7	5	L06	P02	1.00	1.0	2.05	6.0	.34
8	5	L06	P03	1.00	5.0	2.05	6.0	.34
9	5	L07	P03	0.75	5.0	2.05	6.0	.34

Table 5. Pooled LTVs for cross collateralized loans.

COMPUTING RISK MEASURES FOR COMMITMENTS WITH TAKEDOWN LOANS

In commercial lending, lending facilities include current loans and future commitments. A current loan may be a standalone loan or a takedown loan from a future commitment. For a future commitment, several limit lines may be set up by the lender. In such scenarios, hierarchical loan structures are created. Figure 2 shows some examples of loan structures. In case (a), there is only a single takedown loan from a main commitment; in case (b), there are two takedown loans withdrawing from a limit commitment which is a line of credit from a main commitment; in case (c), both limit and sublimit lines of commitments have been set up by the lender. Each line can finance takedown loans not exceeding its line limit. The risk measures of future commitments are usually reported at the highest advised level.

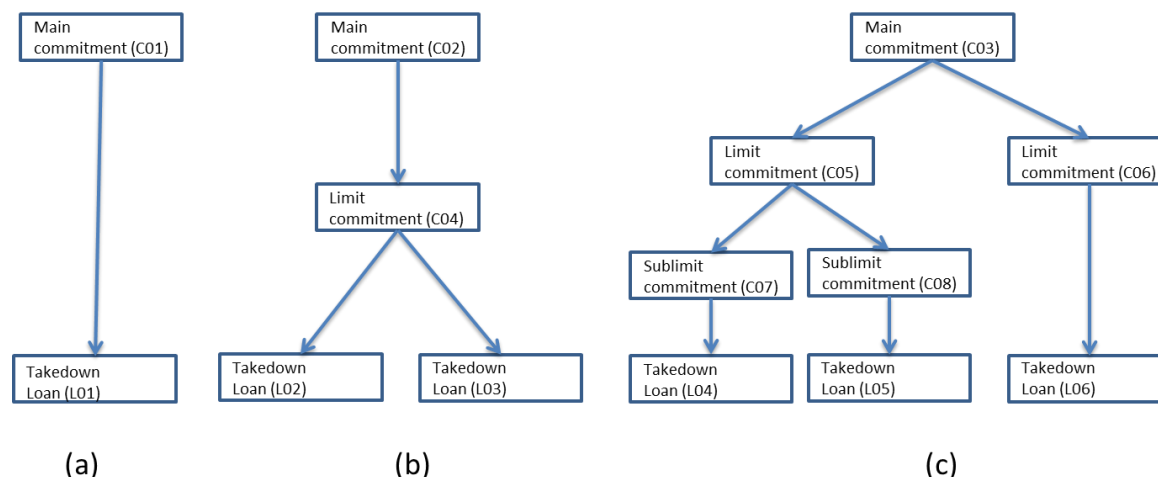


Figure 2. Loan structures with future commitments and takedown loans.

These loan structures are stored in the banks' loan and commitment tables with each obligation having a row entry. In such a table, let *AccountID* identify the child obligation which is either a loan (*Loan_ID*) or a commitment (*Commitment_ID*), and let *parentAccountID* identify the parent obligation from which the child obligation takes line of credit. The hierarchical loan structures can be stored in a SAS data set. A hierarchical structure is a special type of graph with a unidirectional connection. Again, we can use the Union-Find algorithm to find all these hierarchical structures from a SAS data set.

The following code is used to import the loan structures in Figure 2 into a SAS data set:

```

data loantable;
input AccountID $1-3 parentAccountID $5-7 Credit_Limit 9-11 Disbursed_Amount_t0 13-15
      Disbursed_Amount_t1 17-19 Outstanding_Balance_t0 21-22 Outstanding_Balance_t1
      24-26;
datalines;
L01 C01 0 0 0 20 10
L02 C04 0 0 0 15 10
L03 C04 0 0 0 25
L04 C07 0 0 0 40 35

```

```

L05 C08    0    0    0    290
L06 C06    0    0    0    35    30
C01      100  30  30  0    0
C02      100  20  50  0    0
C03      500 100 400  0    0
C04 C02    80  20  50  0    0
C05 C03    500  50 350  0    0
C06 C03    300  50  50  0    0
C07 C05    300  50  50  0    0
C08 C05    400  0 300  0    0
;

```

The SAS data set *loantable* includes a few columns that we will use to illustrate the calculations of exposures at default and credit conversion factors at the loan account level. In a bank's loan portfolio, there will be thousands or millions of such rows in the loan and commitment table.

To calculate risk exposures at the highest level of future commitment, we need to find all hierarchical loan structures from the loan/commitment table and identify the highest level of future commitment. The graph algorithm and macro program presented in the previous section can be applied to solve this problem. We first create the edge list *linkednodelist* from the loan/commitment table, and then using *linkednodelist* as the input, we invoke the macro function *%group_connected_components()*. Here, we set *hierarchy=TRUE* to preserve the hierarchical structure of all connect components.

```

data linkednodelist;
    set loantable (keep=AccountID parentAccountID);
    where parentAccountID is not missing;
run;

%group_connected_components(edgelist=linkednodelist, vx=AccountID, vy=parentAccountID,
hierarchy=TRUE)
proc print data=vlistout; run;

```

Table 6 is the resulted vertex list output, where each vertex (object) is identified by an *ObjectID*, and each *ObjectID* is uniquely identified by a *NodeID*. The *ConcompID* is the *NodeID* of the root of each loan structure. Note that three hierarchical loan structures are discovered: {C01, L01}, {C02, C04, L02, L03}, and {C03, C05, C06, C07, C08, L04, L05, L06}. These disjoint sets are identified by *ConcompID* 1, 2, and 3, respectively.

Obs	ObjectID	NodeID	ConcompID
1	C01	1	1
2	C02	2	2
3	C03	3	3
4	C04	4	2
5	C05	5	3
6	C06	6	3
7	C07	7	3
8	C08	8	3
9	L01	9	1
10	L02	10	2
11	L03	11	2
12	L04	12	3
13	L05	13	3
14	L06	14	3

Table 6. The vertex list output.

Assuming all commitments in the data set *loantable* are in default status and the data have been collected within a data collection window, typically a 12-month period, starting at time t_0 and ending at the default time t_1 . For a main obligation, the actual exposure at default (EAD) is the sum of outstanding balances of all takedown loans under the main obligation at the default time:

$$EAD = \text{sum}(\text{Outstanding_Balance}_{t1}).$$

The credit conversion factor (CCF) can be computed with the following formula:

$$CCF = [EAD - \text{sum}(\text{Outstanding_Balance}_{t0})] / \text{Unused_Commitment_Amount}_{t0},$$

where $\text{Unused_Commitment_Amount}_{t0} = \text{Credit_Limit} - \text{Disbursed_Amount}_{t0}$, and $\text{Disbursed_Amount}_{t0}$ is the disbursed amount at the start of data collection window and Credit_Limit is the advised credit limit of a commitment at the start of data collection window (Brown, 2014; Yang and Tkachenko, 2014). After all main obligations and their

loan structures have been identified, the computations of EAD and CCF for a future commitment are straightforward. The following SAS code is used to accomplish the task. The results are listed in Table 7.

```
proc sql noprint;
create table loantable2 as
select a.*,
       sum(a.Outstanding_Balance_t1) as groupOS_t1,
       sum(a.Outstanding_Balance_t0) as groupOS_t0,
       b.ConcompID
from loantable as a
left join vlistout as b
on a.AccountID=b.ObjectID
group by ConcompID;
quit;

data loantable3;
set loantable2;
if parentAccountID=' ' or substr(AccountID, 1, 1)='L';
if substr(AccountID, 1, 1)='L' then do;
    EAD=Outstanding_Balance_t1;
    CCF=0;
end;
else do;
    EAD=groupOS_t1;
    CCF=(EAD-groupOS_t0)/(Credit_Limit-Disbursed_Amount_t0);
end;
if CCF ne . and CCF<0 then CCF=0;
if CCF ne . and CCF>1 then CCF=1;
drop ConcompID;
run;

proc sort data=loantable3; by AccountID; run;
proc print data=loantable3; run;
```

Obs	AccountID	Parent AccountID	Credit_ Limit	Disbursed_ Amount_t0	Disbursed_ Amount_t1	Outstanding_ Balance_t0	Outstanding_ Balance_t1	EAD	CCF
1	C01		100	30	30	0	0	10	0.00
2	C02		100	20	50	0	0	35	0.25
3	C03		500	100	400	0	0	355	0.70
4	L01	C01	0	0	0	20	10	10	0.00
5	L02	C04	0	0	0	15	10	10	0.00
6	L03	C04	0	0	0	.	25	25	0.00
7	L04	C07	0	0	0	40	35	35	0.00
8	L05	C08	0	0	0	.	290	290	0.00
9	L06	C06	0	0	0	35	30	30	0.00

Table 7. Loan/Commitment table with actual EAD and CCF.

COMPUTING RISK MEASURES FOR CROSS COLLATERALIZED COMMITMENTS

As shown in Figure 2, a main commitment may have several levels of limit and sublimit commitments with takedown loans. In a loan accounting system, collaterals may be recorded at any levels of a deal structure, and the same collateral may be used to secure other loans or commitments that are from separate deal structures. Thus an even more complicated network with cross collateralized loans, commitment, and properties are formed from these business scenarios. The risk measures for loans and commitments in such a cross collateralized hierarchical deal structure are better evaluated if we can place the connected loans, commitments, and properties in the same pool and compute the aggregated risk measures for each pool. The graph algorithm and macro program discussed above can be applied again to find all connected components in this business scenario.

For a future commitment, the loan-to-value ratio can be defined as the ratio of the exposure amount of the commitment to the current value of the collateralized property. With a conservative estimation, the risk exposure of a commitment is equal to the summation of outstanding balance plus unused commitment amount. The LTV ratio of a cross collateralized commitment can be computed in the same way as it is used for cross collateralized standalone loans in previous sections. In addition, other risk measures, such as net operating income, debit service, and debit service coverage ratio, may also be computed for cross-collateralized loans and commitments with the same approach.

CONCLUSION

In summary, I have introduced a SAS macro program that can be used to identify the connected components from a SAS data set and group all connected components into discrete groups. I used simple examples to demonstrate how to apply the macro program to find cross collateralized loans and to compute loan-to-value ratios for these complicated loans. I have also applied the macro program to identify main obligations and loan structures for future commitments with takedown loans and limit/sublimit lines of credit. The computations of exposure at default and credit conversion factor have been illustrated with examples. The algorithm and the macro program presented in this paper are well scalable. It can be used to find connected components in a relational table with thousands to millions of rows with practically linear processing time. The application of this macro program is not just limited to the problem discussed herein. It may be extended to solve other problems that involve hierarchies, trees, networks, and connected components.

REFERENCES

1. Weiss, M. A. (1994). *Data Structures and Algorithm Analysis in C++*. Menlo Park, CA: Addison-Wesley Publishing Company.
2. Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1997). *Introduction to Algorithms*. Cambridge, MA: The MIT Press.
3. Cai, C. (2015). "Implementing Union-Find Algorithm with BASE SAS DATA Steps and Macro Functions". PharmaSUG 2015 Conference proceedings. Available at <http://www.pharmasug.org/proceedings/2015/BB/PharmaSUG-2015-BB06.pdf>.
4. Office of the Comptroller of the Currency. (2013). "Comptroller's Handbook: Commercial Real Estate Lending". Washington, DC. Available at <http://www.occ.gov/publications/publications-by-type/comptrollers-handbook/cre.pdf>.
5. Ono, A., Uchida, H., Udell, G., & Uesugi, I. (2013). "Lending Pro-Cyclicality and Macro-Prudential Policy: Evidence from Japanese LTV Ratios". Available at https://www.hhs.se/contentassets/08e6b51574604199a7d21f870f1ed709/udell_ono-uchida-uesugi.pdf.
6. Brown, I. L. J. (2014). *Developing Credit Risk Models Using SAS® Enterprise Miner™ and SAS/STAT®: Theory and Applications*. Cary, NC: SAS Institute Inc.
7. Yang, B. H., Tkachenko, M. (2014). "Modeling of EAD and LGD: Empirical Approaches and Technical Implementation". Available at <http://mpira.ub.uni-muenchen.de/57298/>.

CONTACT INFORMATION

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

Name: Chaoxian Cai, Ph.D.
 Enterprise: Automated Financial Systems, Inc.
 Address: 123 Summit Drive
 City, State ZIP: Exton, PA 19341
 E-mail: cai.charles.x@gmail.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 A

The following is the SAS macro program discussed in the context:

```
%macro group_connected_components(edgelist=, vx=, vy=, elistout=elistout,
vlistout=vlistout, hierarchy=FALSE);
/*****
Program: group_connected_components.sas
Purpose: For a given undirected graph G(V, E), find all connected components
Author: Chaoxian Cai, 5/15/2016
*****/
```

Parameters:

edgelist: The input edge list data set with edges defined by columns vx and vy
 vx: Left vertex of an edge; for hierarchical tree structure, the child node
 vy: Right vertex of an edge; for hierarchical tree structure, the parent node

hierarchy: Whether to preserve hierarchical structures in the connected components, TRUE or FALSE. Default is set to FALSE.
 elistout: The output edge list data set with each connected component labeled by a unique identification number (ConcompID). Default data set name: elistout.
 vlistout: The output vertex list data set with each vertex (object) identified by ObjectID, NodeID, and ConcompID. Default data set name: vlistout.

Notes:

Union-Find algorithm is implemented to find the connected components. By default, union by depth optimization is applied to shorten tree depth. As a result, the connected components do not preserve hierarchical tree structures; If hierarchy=TRUE is set, then union by depth is not applied. The union of two disjoint sets preserves child-parent hierarchy, and the resulted connected components have hierarchical tree structures. The connected components are identified by their root node IDs.

*****/

```
/* create vertex list from given edge list */
proc sql noprint;
create table vlist as
select &vx as ObjectID
from &edgelist
union
select &vy as ObjectID
from &edgelist
order by ObjectID;
quit;

/* label each vertex sequentially from 1 to N */
data vlist;
  set vlist end=last;
  NodeID + 1;
  if last then call symputx('nNode', NodeID);
run;

%put nNode=&nNode;

/* attach labels to each node in the edge list */
proc sql noprint;
create table elist as
select a.*,
       b.NodeID as leftnode,
       c.NodeID as rightnode
from &edgelist as a
left join vlist as b
on a.&vx=b.ObjectID
left join vlist as c
on a.&vy=c.ObjectID;
quit;

/* apply Union-Find algorithm */
data ConcompID (keep=NodeID ConcompID);
array p[&nNode] _temporary_;
set elist (keep=leftnode rightnode
          rename=(leftnode=x rightnode=y)) end=last;
if _n_ =1 then do;
  do i=1 to &nNode;
    p[i]=0;
  end;
end;

/* find current root of node x */
do while (p[x]>0);
  x=p[x];
end;
```

```

/* find current root of node y */
do while (p[y]>0);
    y=p[y];
end;

/* union subtrees of x and y */
%if %upcase(&hierarchy)=TRUE %then %do;
if (x ne y) then p[x]=y;
%end;
%else %do;
if (x ne y) then do;
    if (p[y]<p[x]) then p[x]=y;
    else do;
        if (p[y]=p[x]) then p[x]=p[x]-1;
        p[y]=x;
    end;
end;
%end;

/* output connected components */
if last then do;
    do i=1 to &nNode;
        NodeID=i;
        root=i;
        do while (p[root]>0);
            root=p[root];
        end;
        ConcompID=root;
        output ConcompID;
    end;
end;
run;

/* create vertex list output data set */
data &vlistout (keep=ObjectID NodeID ConcompID);
merge vlist (in=a) ConcompID;
by NodeID;
if a;
run;

/* create edge list output data set */
data &elistout;
if 0 then set &vlistout (keep=ObjectID ConcompID);
if _N_=1 then do;
    declare hash gidhash (dataset: "&vlistout");
    gidhash.definekey("ObjectID");
    gidhash.definedata("ConcompID");
    gidhash.definedone();
end;

set &edgelist;
if gidhash.find(key: &vx)=0 then do;
end;

drop ObjectID;
run;

%mend group_connected_components;

```