PharmaSUG 2015 - Paper BB06

Implementing Union-Find Algorithm with Base SAS DATA Steps and Macro Functions

Chaoxian Cai, AFS, Exton, PA

ABSTRACT

Union-Find algorithm is a classic algorithm used to form the union of disjoined sets and find connected components of a graph from a given set of vertices and edges. The algorithm is often used in data manipulations that involve graphs, trees, hierarchies, and linked networks. A tree data structure can be used to implement the algorithm. A SAS data set is a tuple data structure that is excellent for table manipulations, but cannot be directly applied for implementing Union-Find algorithm. In this paper, we will discuss the programming techniques that are available in Base SAS for the implementation of the Union-Find algorithm. We will explain how to implicitly represent graphs and trees using SAS arrays, and how to build hierarchical trees and perform queries on trees using SAS DATA steps and macro functions. We will also discuss the programming techniques that have been used to minimize average running time of the algorithm.

INTRODUCTION

Complex networks play important roles in many fields of pharmaceutical research and development. Examples of such network applications include protein-protein interactions, metabolic and biochemical pathways, drug-drug and drug-protein interactions, and so on. A complex network is naturally represented by a graph, which uses vertices to represent individual objects and edges to represent the relationships among individual elements. The most commonly graphs include trees, hierarchies, and linked networks. In programming languages such as C/C++ and Java, the data structure of a graph is typically represented by either an adjacent linked list or an adjacent matrix. Vertices and edges in a graph are also commonly referred as nodes and paths. In a relational database management system (RDBMS), a graph is represented by columns, rows, and relational tuples. 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 row record for each object and add the key of the second object as an attribute of the first object for all connections among the objects of interest. Thus, complex networks can be conveniently stored in a relation table such as a SAS data set. However, performing queries efficiently on a graphical structure in a relational table appears to be challenging.

Performing a SQL query on a graphical data structure often involves writing recursive SQL procedures and requires significant programming efforts. Oracle SQL has implemented a CONNECT BY function that can be used to query hierarchies in a table (Oracle, 2015). SQL Server has provided HIERARCHYID data type and common table expressions (CTE), which can be used to query hierarchies and trees in a table (Ben-Gan et al., 2009). SAS PROC SQL supports recursive joins that can be programmed to query hierarchies in a SAS data set (SAS Institute Inc., 2015). However, even with these functionalities, it is never a trivial task to query, maintain, and manipulate graphs, trees, and hierarches in a RDBMS using SQL procedures. The recursive joins performed by SQL procedures are usually not efficient. Using SQL procedures to find all connected components in a complex network is even more challenging than usual query tasks. In this paper, we will discuss the Union-Find algorithm and its implementation with Base SAS DATA steps and macro functions. Through this implementation, we will demonstrate an elegant solution to find and group connected components in complex networks stored in a SAS data set.

EXAMPLE

To illustrate the problem, first let us introduce a simple example in Table 1, where we are trying to find all linked components of drug-protein interactions. For convenience to discuss the Union-Find algorithm in the subsequent sections, Table 1 contains only two columns: *drug_id* and *protein_id*, which are the primary keys of drugs and proteins, respectively, in a relational database. We have omitted other attributes that are usually associated with an object in a relational table. The *drug_id* number can be either a simple key or a concatenation of a compound key, so does the *protein_id* number. In Table 1, there are 10 drugs and 18 proteins of interest. Each row in this table represents an interaction or a link between a particular drug and a particular protein. There are 13 such links in total.

Our goal is to identify all drug and protein interaction networks from an input SAS data set that will be created from Table 1. In the terminology of graph theory, we need to find all connected components of a graph from a given set of vertices and edges. Here, the vertices are drugs and proteins, and the edges are the ordered pairs of drug and protein interactions.

drug_id	*protein_id*
1	101
1	102
2	101
3	102
4	103
5	103
5	104
6	104
7	105
8	105
9	106
9	107
10	108

Table 1. An example of drug-protein interactions.

Figure 1 shows the expected outcome. The data set in Table 1 only consists of 18 vertices (or nodes) and 13 edges (or links), and by tracing each row in the table visually, we can easily identify 5 connected components ({1, 2, 3, 101, 102}, {4, 5, 6, 103, 104}, {7, 8, 105}, {9, 106, 107}, {10, 108}. However, in reality, a SAS data set which carries network information often consists of thousands and millions of nodes and links; finding and grouping all connected components in a large data set is computationally demanding.

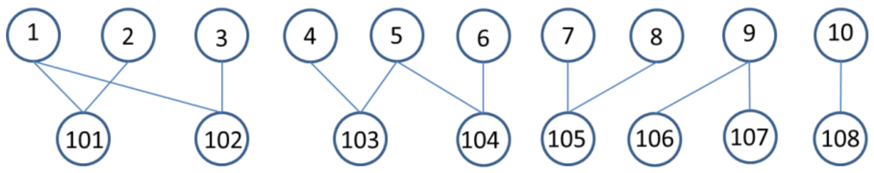

Figure 1. Five connected components in Table 1.

Union-Find algorithm is a classic algorithm used to join disjoined sets and find connected components of a graph from a given set of vertices and edges (Weiss, 1994; Cormen, 1997). In the next sections, we will discuss the Union-Find algorithm and the techniques to implement the algorithm in Base SAS. We will illustrate the SAS implementation step by step using the above simple example, but the solution and the SAS programs present herein should be applicable to complex graphs and networks with large sizes.

DISJOINT SET DATA STRUCTURE

The Union-Find algorithm performs grouping of *N* distinct elements into a collection of disjoint sets. A disjoint set data structure maintains a collection set *S* with *k* number of disjoint sets {S_1, S_2, …, S_k} dynamically. For instance, in the above example in Table 1, the initial disjoint sets are {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}, {9}, {10}, {101}, {102}, {103}, {104}, {105}, {106}, {107}, and {108}. For all observations in Table 1, the algorithm processes each observation exactly once until all observations have been processed, just like how a SAS DATA step iteratively processes a data set. Let $S(x_i)$ and $S(x_j)$ denote the sets that contain elements x_i and x_j, respectively. For *N* distinct elements of x_1, x_2, …, x_i, …, and x_n, there are three basic set operations, namely, *MakeSet*(x_i), *UnionSet*(x_i, x_j), and *FindSet*(x_i). *MakeSet*(x_i) creates a new set $S(x_i)$ that contains x_i; *UnionSet*(x_i, x_j) merges two sets $S(x_i)$ and $S(x_j)$ into one combined set $S(S(x_i), S(x_j))$; *FindSet*(x_i) returns the set $S(x_i)$ that contains x_i.

The disjoint sets in the Union-Find algorithm can be represented by tree data structures. For example, the 5 connected components in Figure 1 are 5 disjoint sets which can be represented with 5 tree data structures (Figure 2). Each set is represented by a tree. Since each element in a disjoint tree shares the same root, each tree can be labeled and identified by its root. In Figure 2, there are 5 trees, which represent five disjoint sets: {1, 2, 3, 101, 102}, {4, 5, 6, 103, 104}, {7, 8, 105}, {9, 106, 107}, and {10, 108}. For a set of *N* distinct elements, initially, there are *N* distinct trees with each tree having only one element at the root.

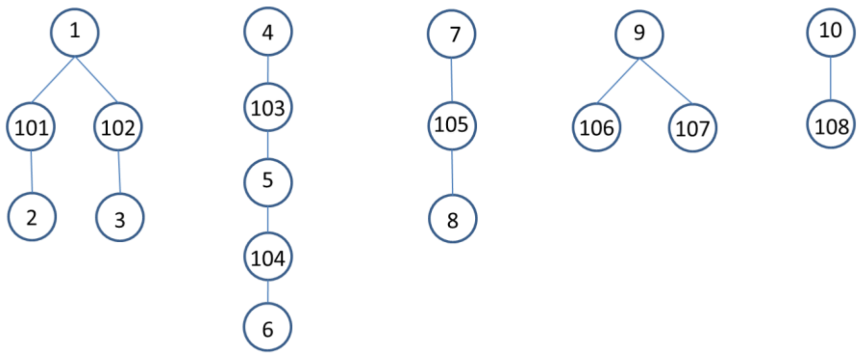

Figure 2. Tree representations of disjoint sets.

A tree is a special type of graph that can be represented by either an adjacent link list or an adjacent matrix. Nevertheless, it is not convenient to implement an adjacent link list in Base SAS. An adjacent matrix may be implemented in a SAS DATA step, but it will require *$O(N^2)$* of memory storage for an input size *N*, which will quickly become impracticable when the size of *N* is getting larger. An alternative to represent a tree data structure is to use a one-dimensional array with size *N*. Since we are only interested in the locations of elements in the tree, we can number each element in all disjointed sets from 1 to *N* sequentially and place each element into the *N*-sized array correspondingly. Thus we can just use the array index to locate each element in the tree. We can represent the tree hierarchy by setting the value of an array element to be the index number of its parent. In this way, a tree has been implicitly represented by a one-dimensional array with an array element representing a node in a tree, an array index identifying the node, and the value of an array element pointing to its parent node.

As illustrated in Figure 3, a data set with *N* objects has been placed into an array *p* of size *N* (*N*=18 in the example). Each array element represents an object that is located by its corresponding index. The value of each entry *p[i]* represents the parent element. If an element is located at the root, we set the value *p[i]* equal to 0 to indicate that it is a root.

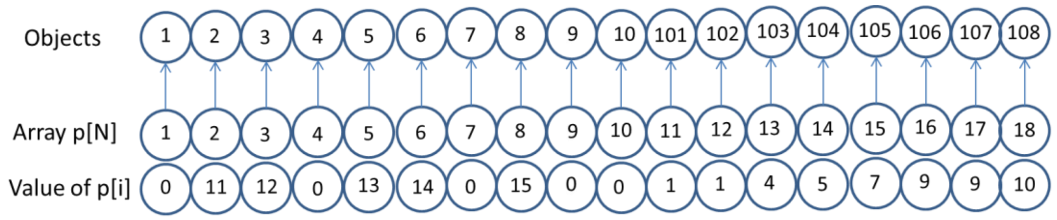

Figure 3. An array representation of the five disjoint trees in Figure 2.

Let 1, 2, 3, …, 17, 18 be the indices of an array that points to the original 18 objects in the given example. Using the array indices as the labels for the original object, we have 5 disjoint sets: {1, 2, 3, 11, 12}, {4, 5, 6, 13, 14}, {7, 8, 15}, {9, 16, 17}, and {10, 18}. Each disjoint set can be identified by its root index: 1, 4, 7, 9, and 10, respectively. All elements in a disjoint set can be found by tracing back each element to its parent until to the root. For example, starting from the element at *i*=6, if we trace the parent node of element 6 recursively, we have *p[6]*=14, *p[14]*=5, *p[5]*=13, *p[13]*=4, and *p[4]*=0, where 0 indicating the root of a disjoint set.

DETAILS OF THE UNION-FIND ALGORITHM

In this section, we will discuss the details of Union-Find algorithm and implement the algorithm using SAS macro functions. Assuming there are *N* objects initially, we will place these *N* objects into *N* macro variables sequentially. Thus, the *N*-sized one-dimensional array is defined by *N* macro variables that are named sequentially. The following macro function *%MakeSet(x)* declares a macro variable *p&x* and sets its value to 0, which represents a single node tree with element *x* at the root. The macro function *%InitDisjointSets(N)* creates an array of *N* macro variables *p&i*, and initialize all values of its elements to 0. A single step *MakeSet(x)* operation takes *O(1)* running time, and the initialization of *N* disjoint sets takes *O(N)* running time.

```
%macro MakeSet(x);
    %global p&x;
    %let p&x=0;
%mend;

%macro InitDisjointSets(N);
    %do i=1 %to &N;
    %MakeSet(&i)
    %end;
%mend;
```

The operation of *FindSet(x)* traces the parent of node *x* recursively from *x* to the root. We write the following macro function *%FindSet(x)* to do the task. This is a recursive macro function that calls itself until the root is found and returned. This step takes *O(d)* running time, where *d* is the depth of the tree.

```
%macro FindSet(x);
    %if &&p&x<=0 %then &x;
    %else %FindSet(&&p&x);
%mend;
```

The macro function *%FindSet(x)* can also be implemented if we use a %do %while loop to replace the tail recursion. In the following code, the parent of node *p&x* is updated inside a %do %while loop until the root is found and returned.

```
%macro FindSet(x);
    %do %while (&&p&x>0);
        %let x=&&p&x;
    %end;
    &x
%mend;
```

The operation of *UnionSet(x, y)* merges two disjoint sets into one new set by making the root of one disjoint set a child of the root of the other disjoint set. The algorithm first calls *FindSet(x)* and *FindSet(y)* to find the root node of the tree that contains *x* and *y*, respectively. If *FindSet(x)* and *FindSet(y)* return the same root, *x* and *y* are already in the same set. If they return different root, we create a new disjoint set by placing one tree as a sub tree under the root of the other tree. The operation is illustrated in Figure 4.

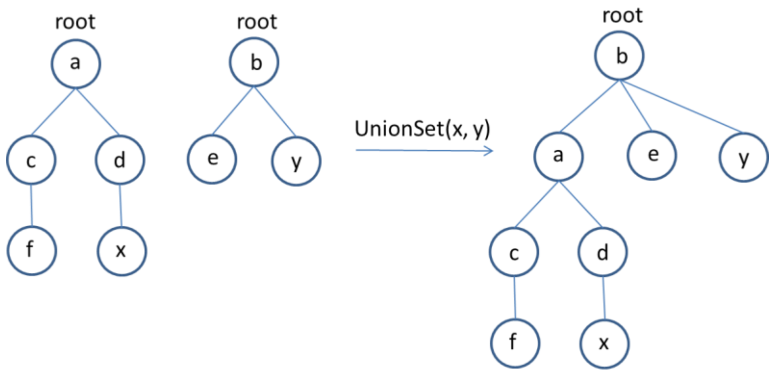

Figure 4. A single *UnionSet(x,y)* operation.

The macro function *%UnionSet(x, y)* is implemented as follows:

```
%macro UnionSet(x, y);
    %let a = %FndSet(&x);
    %let b = %FindSet(&y);
```

```
        %if (&a ne &b) %then %do;
            %let p&b=&a;
        %end;
    %mend;
```

The above union operation is performed arbitrarily in that the first tree is made a sub tree of the second tree. Without further optimization, this arbitrary tree merge may produce a degenerated linear tree at worst case, which has a tree depth of *N-1* from the furthest leaf to its root for a tree with *N* nodes. Thus, the worst case of a *UnionSet(x, y)* operation without any optimization may cost *O(N)* running time. This is undesirable. Fortunately, there are two simple techniques to improve the union operation, namely, Union by Rank and Union by Size. Both techniques will guarantee to produce a tree with depth of at most *O(log(N))* for *N* numbers of tree nodes. In both techniques, the strategy is making the root of the smaller tree a child of the root of the bigger tree. In Union by Rank, two trees are merged based on the height of each tree; similarly, in Union by Size, two trees are merged based on the size of each tree.

In Union by Rank operation, we keep track of tree depths when the trees are growing. We set the depth of a tree root to 0, the depth of the nodes one level below the root to -1, the depth of the nodes two level below the root to -2, and so on. We save the depth of a tree at its root with the most negative depth from the furthest leaf to the root. When two trees with unequal depth are merged, the tree that is shallower becomes a child of the root of the deeper tree, and the depth of the merged tree is unchanged. When two trees with equal depth are merged, we will break the tie arbitrarily. We simply choose the root of one tree as the parent and place the root of the other tree as the child. In this case, we need to update the depth of the newly merged tree by decrementing its depth by 1, one level deeper in the negative direction. Figure 5 shows an example of a single *UnionSet(x, y)* operation if Union by Rank technique is applied. With Union by Rank, the *UnionSet(x, y*) step takes at most *O(log(N))* running time in worst case.

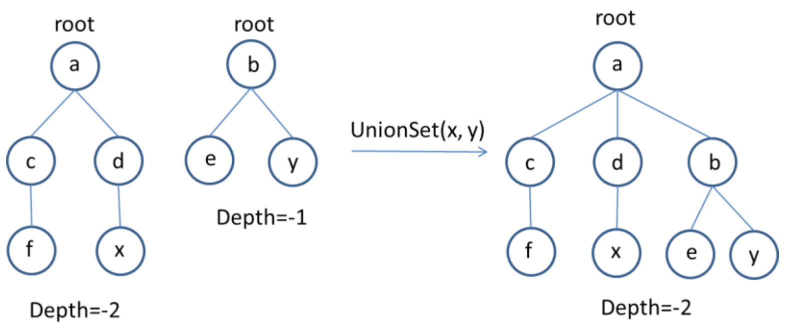

Figure 5. A Union by Rank operation.

With the incorporation of Union by Rank technique, the previous *%UnionSet(x, y)* macro function is modified as follows:

```
    %macro UnionSet(x, y);
        %let a = %FindSet(&x);
        %let b = %FindSet(&y);
        %if (&a ne &b) %then %do;
            %if (&&p&b < &&p&a) %then %do;
                %let p&a=&b;
            %end;
            %else %do;
                 %if (&&p&b = &&p&a) %then %do;
                     %let p&a=%eval(&&p&a - 1);
                 %end;
                %let p&b=&a;
            %end;
        %end;
    %mend;
```

In addition to the running time improvement in *UnionSet(x, y)* operation, we can improve *FindSet(x)* operation by implementing a technique called Path Compression. That is, when running *FindSet(x)*, we make every node on the path from *x* to the root have its parent changed to the root. Such an example is shown in Figure 6.

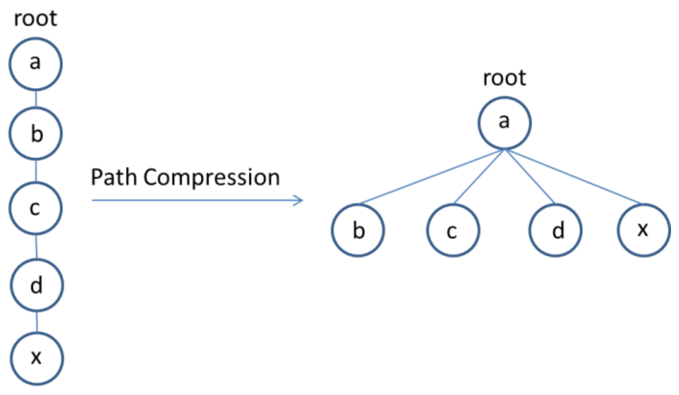

Figure 6. A Path Compression example.

The following modified SAS code implemented the Path Compression technique in the *FindSet(x)* step. For node *x*, the macro function *%FindSet_And_Compress(x)* first find the root of *x* iteratively, and then set the parents to root for all nodes in the path from *x* to the root.

```
%macro FindSet_And_Compress(x);
    %let i=&x;
    %do %while (&&p&x>0);
        %let x=&&p&x;
    %end;
    %let root=&x;
    /* set parent to root for all nodes in the path from x to root */
    %do %while (&&p&i>0);
         %let j=&i;
         %let i=&&p&i;
            %let p&j=&root;
    %end;
    &root
%mend;
```

The Path Compression technique greatly flattens all disjoint set trees and shortens the tree depths. We have tested the algorithm using a simulated data set of size 100,000 of which 30% nodes are randomly linked, and we found that the tree depths were never more than 3 levels below the roots. If both Union by Rank and Path Compression techniques are implemented in the Union-Find algorithm, it can be shown that the average running time of the algorithm is nearly linear (Weiss, 1994; Cormen, 1997).

STEP BY STEP WORKFLOWS

Putting all together, we implemented the following stepwise workflow and SAS code to find all drug-protein interactions in Table 1 using the Union-Find algorithm.

Step 1: Input the given example and create an input SAS data set.

```
data example;
    input drug_id protein_id @@;
    datalines;
1 101 1 102 2 101 3 102 4 103 5 103 5 104 6 104 7 105 8 105 9 106 9 107 10 108
;
```

Step 2: Output the keys of all objects of interest in the input data set to a new data set that will have a single column of keys only. Sort this data set, and remove duplicates. Count the total number of observations (*N*) in this data set, and number each key sequentially from 1 to *N* by using the SAS system variable *_N_*. Output the sequential number *_N_* as *ID* along with its original key to another new data set, which will serve as a lookup table in next steps. We will use the sequential number to find the original object in this lookup table. The sequential numbers also serve as the indices to the *N* disjoint sets in an array that we will create in a later step.

```
data example_keys;
    set example (keep=drug_id rename=(drug_id=key))
        example (keep=protein_id rename=(protein_id=key));
run;

proc sort data=example_keys nodupkey; by key; run;

data example_ids;
    set example_keys end=last;
    ID=_N_;
    if last then call symputx('N', _N_);
run;
```

Step 3: Merge the sequential *ID* numbers back to the original input data set for *drug_id* and *protein_id*, respectively. Rename *ID* for *drug_id* as *drugID*, and *ID* for *protein_id* as *proteinID*. The data set created in this step will feed directly to the *UnionSet(x, y)* operations in Step 5, where the *x* and *y* parameters are *drugID* and *proteinID*, respectively.

```
proc sql noprint;
    create table example4ufinput as
    select a.*,
           b.ID as drugID,
           c.ID as proteinID
    from example as a
         left join example_ids as b
         on a.drug_id = b.key
         left join example_ids as c
         on a.protein_id = c.key;
quit;
```

Step 4: Initialize the original *N* disjoint sets by invoking the macro function *%InitDisjointSets(N)*. In this step, each set contains only a single element at the root with all *p[i]*=0 (or *p&i*=0 if array *p* is represented by macro variables).

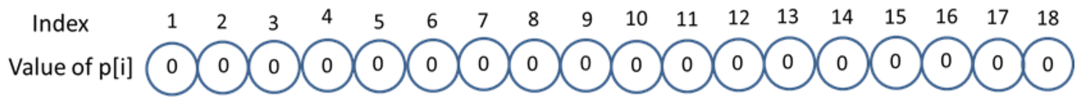

Figure 7. Array values after initialization.

Step 5: Process the input data set by invoking the macro function *%UnionSet(x, y)*, where the *x* and *y* parameters are *drugID* and *proteinID*, respectively. The calls of *%UnionSet(drugID, proteinID)* are dynamically generated after *CALL EXECUTE* routine is repeatedly executed in the DATA step.

```
data _null_;
    set example4ufinput (keep=drugID proteinID);
    call execute('%UnionSet(' || drugID || ', ' || proteinID || ')');
run;
```

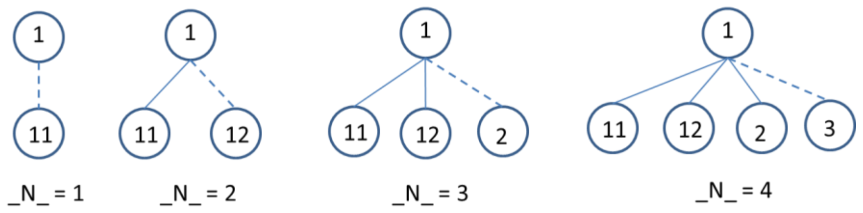

Figure 8. Stepwise unions of 2 disjoint sets during the first 4 iterations.

Figure 8 shows the stepwise unions of 2 disjoint sets during the first 4 DATA step iterations. The dot lines represent the union paths in the formation. Each observation in the input table is processed only once, and the disjoint set trees are built sequentially. In the first iteration (_N_=1), sets {1} and {11} form the union; in the second step (_N_=2), sets {1, 11} and {12} form the union, and the process continues until all observations are processed. Figures 9-12 show the updated sequences of the array values after the first 4 iterations. An array value of negative or zero indicates the root of a disjoint tree; a positive array value represents the location of the parent node in the array. After the first 4 observations are processed, the first connect component is built, and the disjoint set is {1, 2, 3, 11, 12}. This disjoint set tree is rooted at element 1 with a tree depth of -1.

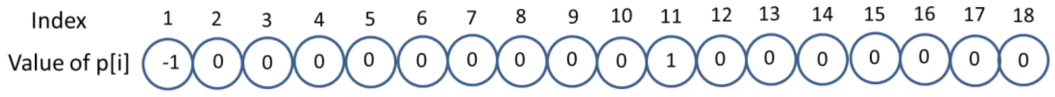

Figure 9. Array values after the 1st observation.

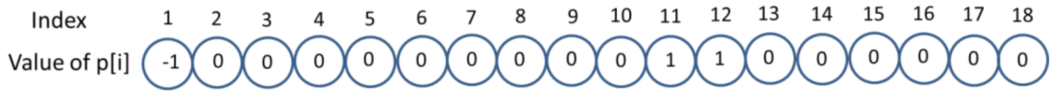

Figure 10. Array values after the 2nd observation.

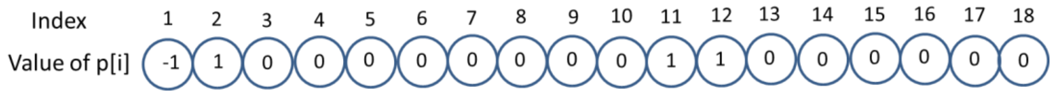

Figure 11. Array values after the 3rd observation.

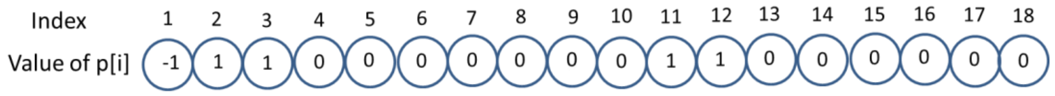

Figure 12. Array values after the 4th observation.

After all observations are processed, all connected elements have been merged and all disjoint sets have been built. The root of each disjoint set is indicated by non-positive tree depth, and all child nodes are pointing to their corresponding parents. The 5 disjoint set trees are {1, 2, 3, 11, 12}, {4, 5, 6, 13, 14}, {7, 8, 15}, {9, 16, 17}, and {10, 18}. The roots are 1, 4, 7, 9, and 10, respectively.

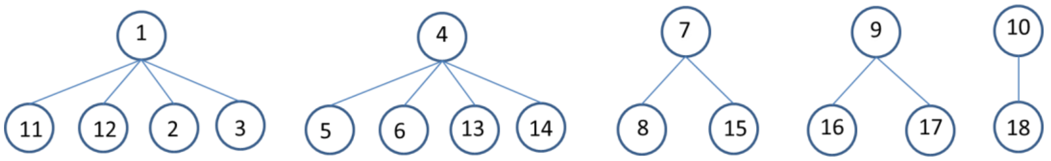

Figure 13. The 5 disjoint trees after all observations are processed.

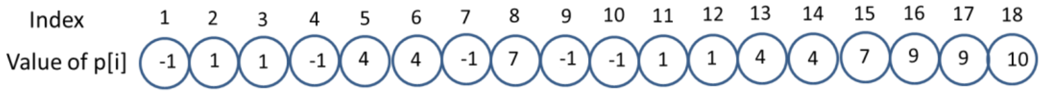

Figure 14. Array values after all observations are processed.

Step 6: Finally, we need to traverse the *N*-sized array and output all disjoint sets to a data set. We invoke the following macro function *%GetDisjointSet* to accomplish the task.

```
%macro GetDisjointSet;
    data disjointsets;
    %local i;
    %do i=1 %to &N;
        ID=&i;
        DisjointSetID=%FindSet(&i);
        output;
    %end;
    run;
%mend;
%GetDisjointSet;
```

Table 2 lists all disjoint sets with *DisjointsetID*, *ID*, and the original object keys (*drug_id* or *protein_id*). Each disjoint set is a distinct connected component that contains all linked objects in the input data set. In another word, every object, whether it is a drug or a protein, have been correctly placed into 5 distinct groups based on their connectivity.

DisjointSetID	***ID***	***drug_id or protein_id***
1	1	1
1	2	2
1	3	3
1	11	101
1	12	102
4	4	4
4	5	5
4	6	6
4	13	103
4	14	104
7	7	7
7	8	8
7	15	105
9	9	9
9	16	106
9	17	107
10	10	10
10	18	108

Table 2. The resulted dataset with 5 connected components.

DATA STEP IMPLEMENTATION

The Union-Find algorithm can also be implemented concisely in a single SAS DATA step. Without resorting to using the macro functions of *%MakeSet(x)*, *%FindSet(x)*, and *%UnionSet(x, y)*, we can integrate these set operations into a single DATA step as follows:

```
data disjointsets (keep=ID disjointsetID);
    array p[&N] _temporary_;
    set example4ufinput (keep=drugID proteinID
                         rename=(drugID=x proteinID=y)) end=last;
    if _n_=1 then do;
```

```
            do i=1 to &N;
                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 (x ne y) then do;
             if (p[y] < p[x]) then do;
                 p[x]=y;
             end;
             else do;
                 if (p[y] = p[x]) then do;
                    p[x] = p[x] - 1;
                 end;
                 p[y]=x;
             end;
         end;

         /** get root for each node **/
         if last then do;
             do i=1 to &N;
                ID=i;
                root=i;
                do while (p[root]>0);
                    root=p[root];
                end;
                disjointsetID=root;
                output disjointsets;
             end;
         end;
     run;
```

In the above code, a temporary array *p* with size *N* is used to represent the original *N* objects in the input data set. When the DATA step reads the first observation, *p[i]* is set to 0 for each array element, representing the original *N* disjoint sets. Instead of invoking macro functions, we integrate the set operations of Union-Find algorithms directly into a DATA step. During each iteration, the first do while loop finds the root for element *x*, and so does the second do while loop for finding the root of element *y*. Next, via Union by Rank operations, the disjoint set containing x merges with the disjoint set that containing y in the way we have discussed previously. After the last iteration, all disjoint sets have been built. We can identify these disjoint sets by looping through each element in the array and outputting the element *ID* and its root *ID* to a data set. We found that the single DATA step implementation is more efficient and performed much faster than the implementation that involving many macro variable definitions and function calls.

CONCLUSION

In conclusion, we have discussed and implemented the Union-Find algorithm using Base SAS DATA steps and macro functions. We have illustrated the algorithm step by step and presented the programming techniques for how to implement the algorithm efficiently in Base SAS. Disjoint set data structure and tree representation are used to implement the algorithm, and the trees are implicitly represented by a linear array. Macro functions have been implemented and explained in detail for making disjoint sets, finding an element in a disjoint set, and merging two disjoint sets. The time complexity of the algorithm has been briefly discussed. Two important techniques, Union by Rank and Path Compression, have been discussed and implemented to improve the average running time.

Finally, a complete working program and workflow have been presented with an illustrative example. Although we illustrated the algorithm with this simple example, the algorithm is useful in other network problems, such as road systems, bill of materials, organization charts, and so on. The algorithm can be used to find disjoint sets, connected networks, and the like problems from relational database tables with nearly linear running time.

REFERENCES

1. Oracle and/or Its Affiliates. “Hierarchical Queries”. *Database SQL Reference*. (2015). Available at http://docs.oracle.com/cd/B19306_01/server.102/b14200/queries003.htm
2. Ben-Gan, I., Kollar, L., Sarka, D., & Kass, S. (2009). *Inside Microsoft SQL Server 2008: T-SQL Querying*. Redmond, WA: Microsoft Press
3. SAS Institute Inc. “Sample 25437: Demonstrates recursive joins with PROC SQL”. *Knowledge Base / Samples & SAS Notes*. (2015). Available at http://support.sas.com/kb/25/437.html
4. Weiss, M. A. (1994). *Data Structures and Algorithm Analysis in C++*. Menlo Park, CA: Addison-Wesley Publishing Company
5. Cormen, T. H., I., Leiserson, C. E., & Rivest, R. L. (1997). *Introduction to Algorithms*. Cambridge, MA: The MIT Press

CONTACT INFORMATION

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

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