Paper 084-2009

Implementing Stack and Queue Data Structures with SAS® Hash Objects

Larry Hoyle, Institute for Policy and Social Research, University of Kansas

ABSTRACT

The SAS hash object is a convenient tool for implementing two common data structures, the stack and the queue. While either of these may be implemented with arrays, the hash object implementation offers the advantage of dynamic memory management – a maximum memory size does not need to be specified in advance. This paper includes a set of SAS macros that can be used in a DATA step to create and delete stacks or queues and to enter or remove data from those data structures.

INTRODUCTION

STACKS

A stack is an ordered list of items. Items are added to the list at the top and items are removed from the top. Since the last item added to the list is the first removed from the list, stacks are also known as “Last In First Out” (LIFO) lists. A stack is easily implemented in an array, requiring only a pointer variable to point to the position of the top element of the stack.

QUEUES

A queue is also an ordered list of items, differing from a stack in that the first item added to the queue is the first item to be removed. A queue is also known as a “First In First Out” (FIFO) list. A queue can also be implemented in an array, but managing the pointers to the beginning and end of the array is somewhat more complicated – as the queue runs into the end of the physical array it must wrap around to the beginning of the array to use any space freed by removing items from the queue.

THE HASH OBJECT

A hash object is comparable to an array except that the index into the array, the key, need not be ordered. (In Perl, a hash is known as an associative array). The SAS hash object automatically allocates memory as items are added. In this application the key is a sequential number which, in the case of the queue, can just continue to grow larger as the smallest values are abandoned. Using a key of length 8 should allow 9,007,199,254,740,992 (the largest integer represented exactly) insertions into a queue, which is probably adequate for most applications.

WHY BOTHER?

One reason for using these macros would be to implement a known algorithm that is expressed in terms of stack and queue operations. Checking that a data step correctly implemented that algorithm would be much easier if it were written using push and pop statements or queue and dequeue statements than if those operations were written in terms of conventional DATA step code. Such an example is presented at the end of this paper.

STACK MACROS

The basic operations needed to use a stack are: create the stack, delete the stack, add an item to the stack (push), delete an item from the stack (pop), and check the length of the stack. An additional operation to dump the contents of the stack can be useful for testing. Macros for implementing these operations are listed below.

```
%macro StackDefine(stackName = Stack1,          /*  Name of the stack - use the name in       */
                                                /*       push and pop of this stack           */
                   dataType = n,                /*  Datatype n - numeric                      */
                                                /*           c - character                    */
                   dataLength = 8,              /*  Length of data iitems                     */
                   hashexp = 8,                 /*  Hashexp for the hash - see the            */
                                                /*    documentation for the                   */
                                                /*    hash object. You may want to increase   */
                                                /*    this for a stack that                   */
                                                /*    will get really large                   */
                   rc = Stack1_rc  );           /*  Return code for stack operations          */
                                                /*  the macro will create the following data  */
                                                /*  objects and variables                     */
                          /* &StackName._Hash,       the hash object used for the stack      */
                          /* &StackName._key,        the hash object key variable            */
                          /* &StackName._data,       the hash object data variable           */
                          /* &StackName._end,        variable to hold the number of objects  */
                          /*                         in the stack                            */
```

```
   retain &StackName._end  0;          /*  empty stack has 0 items                          */
   length &StackName._key 8;          /*  key is numeric count of items in the stack       */
                                      /*   making this length 4 would save memory and work */
                                      /*   for a large number of items in the stack        */
   call missing(&StackName._key);     /* explicit assignment so SAS does not complain      */
   %IF &dataType EQ n %THEN %DO;
     length &StackName._data &datalength;
     retain &StackName._data 0;
   %END;
   %ELSE %DO;
     length &StackName._data  $ &datalength;
     retain &StackName._data ' ';
   %END;
   declare hash &StackName._Hash(hashexp: &hashexp);
    &rc = &StackName._Hash.defineKey("&StackName._key");
    &rc = &StackName._Hash.defineData("&StackName._data");
    &rc = &StackName._Hash.defineDone();
                                     /*  ITEM_SIZE attribute available in SAS 9.2
                                        itemSize = &StackName._Hash.ITEM_SIZE; */
  itemSize = 8 + &datalength;
  put "Stack  &StackName. Created. Each Item will take " ItemSize " bytes.";
%mend StackDefine;

%macro StackPush(stackName = Stack1,          /*  Name of the stack -                      */
                 InputData = Stack1_data,     /*  Variable containing value to be pushed   */
                 StackLength = Stack1_length, /*  Returns the length of the stack          */
                 rc = Stack1_rc  );           /*  return code for stack operations         */
  &StackName._end+1 ;                       /*  new item will go in new location in the hash  */
  &StackLength = &StackName._end;
  &StackName._key = &StackName._end;        /* new value goes at the end */
  &StackName._data = &InputData;            /* value from &InputData     */
  &rc = &StackName._Hash.add();
  if &rc ne 0 then put "NOTE: PUSH to stack &stackName failed " &InputData=
                       &StackName._end= ;
%mend StackPush;

%macro StackPop(stackName = Stack1,           /*  Name of the stack -                      */
                OutputData = Stack1_data,     /*  Variable containing value to be pushed   */
                StackLength = Stack1_length,  /*  Returns the length of the stack          */
                rc = Stack1_rc  );            /*  return code for stack operations         */
  if &StackName._end > 0 then do;
    &StackName._key = &StackName._end;          /* return value comes off of the end */
    &rc = &StackName._Hash.find();
    if &rc ne 0 then put "NOTE: POP from stack &stackName could not find "
                         &StackName._end= ;

    &OutputData = &StackName._data;              /* value into &InputData          */
                                                 /*  remove the item from the hash */
    &rc = &StackName._Hash.remove();
    if &rc ne 0 then put "NOTE: POP from stack &stackName could not remove "
                         &StackName._end= ;

    &StackName._end = &StackName._end - 1 ;      /*  stack now has 1 fewer item  */
    &StackLength = &StackName._end;
  end;
  else do;
    &rc = 999999;
    put "NOTE: Cannot pop empty stack  &StackName into &OutputData ";
  end;
%mend StackPop;
%macro StackLength(stackName = Stack1,            /*  Name of the stack -                  */
                   StackLength = Stack1_length, /*  Returns the length of the stack      */
                   rc = Stack1_rc   );          /*  return code for stack operations     */
                   &StackLength = &StackName._end;
%mend StackLength;
```

```
%macro StackDelete(stackName = Stack1,          /*  Name of the stack -                  */
                   rc = Stack1_rc               /*  return code for stack operations     */
                  );
  &rc = &StackName._Hash.delete();
  if &rc ne 0 then put "NOTE: Cannot delete stack  &StackName ";
%mend StackDelete;

%macro StackDump(stackName = Stack1,          /*  Name of the stack -                  */
                 rc = Stack1_rc   );         /*  return code for stack operations     */
  if &StackName._end <= 0 then do;
    put // "Stack &Stackname is empty";
  end;    /*  &StackName._end <= 0 */
  else do;
    put // " Contents of Stack &Stackname:";
    do ixStack = 1 to &StackName._end ;
      &StackName._key = ixStack;
      &rc = &StackName._Hash.find();
      put "item " ixStack  "value "  &StackName._data;
    end; /* do ixStack = 1 to &StackName._end  */
  end;  /* not &StackName._end <= 0     */
%mend StackDump;
```

QUEUE MACROS

The basic operations needed to use a queue are: create the queue, delete the queue, add an item to the queue (enqueue), delete an item from the queue (dequeue), and check the length of the queue. An additional operation to dump the contents of the queue can be useful for testing. Macros for implementing these operations are listed below. The advantage of using a hash object to underlie the queue can be seen in the QueueEnqueue operation where the key for a new item can always be just 1 more than the previous value and in the QueueDequeue operation, where cleaning up after removing an item is just a matter of issuing the remove operation.

```
%macro QueueDefine(QueueName = Queue1,         /*  Name of the Queue -   use in enqueue and */
                                               /*                        dequeue           */
                   dataType = n,               /*  Datatype n - numeric                    */
                                               /*           c - character                  */
                   dataLength = 8,             /*  Length of data items                    */
                   hashexp = 8,                /*  Hashexp for the hash - see the          */
                                               /*   documentation for the  hash object.    */
                                               /*   You may want to increase this for      */
                                               /*    a Queue that will get really large    */
                   rc = Queue1_rc  );          /*  return code for Queue operations        */
                          /*  the macro will create the following data objects and variables */
                           /* &QueueName._Hash,        the hash object used for the Queue    */
                           /* &QueueName._key,         the hash object key variable          */
                           /* &QueueName._data,        the hash object data variable         */
                           /* &QueueName._old,         variable points to the first item put */
                           /*                            in the queue                        */
                           /* &QueueName._new,         variable points to the last item put  */
                           /*                            in the queue                        */
                           /* &QueueName._len,         number of items in the queue          */

   retain &QueueName._new  0;           /*  empty Queue has 0 locations in the hash          */
   retain &QueueName._old  1;           /*  old will be at location 1 when something is added */
   retain &QueueName._len  0;           /*  empty Queue has 0 items                          */
   length &QueueName._key 8;            /*  key is numeric count of items in the Queue       */
   call missing(&QueueName._key);       /* explicit assignment so SAS does not complain      */
   %IF &dataType EQ n %THEN %DO;
     length &QueueName._data &datalength;
     retain &QueueName._data 0;
   %END;
   %ELSE %DO;
     length &QueueName._data  $ &datalength;
     retain &QueueName._data ' ';
   %END;
   declare hash &QueueName._Hash(hashexp: &hashexp);
    &rc = &QueueName._Hash.defineKey("&QueueName._key");
    &rc = &QueueName._Hash.defineData("&QueueName._data");
```

```
    &rc = &QueueName._Hash.defineDone();
                                                    /*  ITEM_SIZE attribute available in SAS 9.2
                                                        itemSize = &QueueName._Hash.ITEM_SIZE; */
  itemSize = 8 + &datalength;
  put "Queue  &QueueName. Created. Each Item will take " ItemSize " bytes.";
%mend QueueDefine;

%macro QueueEnqueue(QueueName = Queue1,         /*  Name of the Queue -                    */
                    InputData = Queue1_data,    /*  Variable containing value to be enqueued */
                    QueueLength = Queue1_length, /*  Returns the length of the Queue       */
                    rc = Queue1_rc   );         /*  return code for Queue operations       */
  &QueueName._new+1 ;                          /*  item goes at new key in hash  */
  &QueueName._len+1 ;                          /*  Queue is 1 longer              */
  &QueueLength = &QueueName._len;
  &QueueName._key = &QueueName._new;           /* new value goes at the end */
  &QueueName._data = &InputData;               /* value from &InputData     */
  &rc = &QueueName._Hash.add();
  if &rc ne 0 then put "NOTE: Enqueue to Queue &QueueName failed " &InputData=
                       &QueueName._new= ;
%mend QueueEnqueue;

%macro QueueDequeue(QueueName = Queue1,          /*  Name of the Queue -                  */
                    OutputData = Queue1_data,    /*  Variable containing value to be dequeued*/
                    QueueLength = Queue1_length, /*  Returns the length of the Queue      */
                    rc = Queue1_rc   );          /*  return code for Queue operations     */
  if &QueueName._len > 0 then do;
    &QueueName._key = &QueueName._old;         /* return value comes from the  oldest location */
                                               /*  in the hash                                 */
    &rc = &QueueName._Hash.find();
    if &rc ne 0 then put "NOTE: Dequeue from Queue &QueueName could not find "
                         &QueueName._new= ;
    &OutputData = &QueueName._data;              /* value into &InputData                  */
    &rc = &QueueName._Hash.remove();             /*  remove the item from the hash */
    if &rc ne 0 then put "NOTE: Dequeue from Queue &QueueName could not remove "
                         &QueueName._new= ;
    &QueueName._old+1 ;                          /*  item comes from oldest location in the hash  */
    &QueueName._len+(-1) ;                       /*  Queue is 1 shorter                           */
    &QueueLength = &QueueName._len;
  end;
  else do;
    &rc = 999999;
    put "NOTE: Cannot Dequeue empty Queue  &QueueName into &OutputData ";
  end;
%mend QueueDequeue;

%macro QueueLength(QueueName = Queue1,          /*  Name of the Queue -               */
                   QueueLength = Queue1_length, /*  Returns the length of the Queue   */
                   rc = Queue1_rc   );          /*  return code for Queue operations  */
  &QueueLength = &QueueName._len;
%mend QueueLength;

%macro QueueDelete(QueueName = Queue1,         /*  Name of the Queue -               */
                   rc = Queue1_rc   );         /*  return code for Queue operations  */
  &rc = &QueueName._Hash.delete();
  if &rc ne 0 then put "NOTE: Cannot delete Queue  &QueueName ";
%mend QueueDelete;

%macro QueueDump(QueueName = Queue1,          /*  Name of the Queue -                */
                 rc = Queue1_rc               /*  return code for Queue operations   */
                 );
  if &QueueName._len <= 0 then do;
    put // "Queue &Queuename is empty";
  end;    /*  &QueueName._end <= 0 */
  else do;
    put // " Contents of Queue &Queuename:";
```

```
    do ixQueue = &QueueName._old to &QueueName._new ;
      &QueueName._key = ixQueue;
      &rc = &QueueName._Hash.find();
      put "item " ixQueue  "value "  &QueueName._data;
    end; /* do ixQueue = QueueName._old to &QueueName._new    */
  end;  /* not &QueueName._end <= 0      */
%mend QueueDump;
```

TESTING

The two sections which follow show test code and the resulting output to the log. The test code creates stack and a queue, checks queue length, adds some items, dumps the data structure, removes some items, and deletes the data structure. Resulting output to the SAS Log is shown in boxes to the right of the code.

TEST CODE - STACK

```
data test;

put /// 'Stack Test' /;
if _n_=1 then do;
  %StackDefine(stackName = testStack,
               dataType = n,
               dataLength = 8,
               hashexp = 8,
               rc = testStack_rc
               );
  put 'define: ' testStack_rc=;
end;
%StackLength(stackName = testStack,
             StackLength = testStack_length,
             rc = testStack_rc );
put / 'length: ' testStack_length = testStack_rc=;

%StackDump(stackName = testStack,
           rc = testStack_rc );


put;
 do myData = 1 to 5;
  %StackPush(stackName = testStack,
                         InputData = myData,
                         StackLength = testStack_length,
                         rc = testStack_rc
             );
put 'push: ' myData= myDataPopped= testStack_length= /
    '      '     testStack_rc= testStack_end=;
end;

%StackDump(stackName = testStack,   rc = testStack_rc );

put;
do ixPop = 1 to 6;
%StackPop(stackName = testStack,
          OutputData = myDataPopped,
          StackLength = testStack_length,
          rc = testStack_rc
         );
put 'pop: ' ixPop= myData= myDataPopped=
            testStack_length= /
    '      ' testStack_rc= testStack_end=;
end;

%StackDump(stackName = testStack,
           rc = testStack_rc );
```

```
Stack Test
```

```
Stack  testStack Created. Each Item will take 16  bytes.
define: testStack_rc=0
```

```
length: testStack_length=0 testStack_rc=0
```

```
Stack testStack is empty
```

```
push: myData=1 myDataPopped=.
testStack_length=1
    testStack_rc=0 testStack_end=1
push: myData=2 myDataPopped=.
testStack_length=2
    testStack_rc=0 testStack_end=2
push: myData=3 myDataPopped=.
testStack_length=3
    testStack_rc=0 testStack_end=3
push: myData=4 myDataPopped=.
testStack_length=4
    testStack_rc=0 testStack_end=4
push: myData=5 myDataPopped=.
testStack_length=5
    testStack_rc=0 testStack_end=5
```

```
Contents of Stack testStack:
item 1 value 1
item 2 value 2
item 3 value 3
item 4 value 4
item 5 value 5
```

```
pop: ixPop=1 myData=6 myDataPopped=5 testStack_length=4
    testStack_rc=0 testStack_end=4
pop: ixPop=2 myData=6 myDataPopped=4 testStack_length=3
    testStack_rc=0 testStack_end=3
pop: ixPop=3 myData=6 myDataPopped=3 testStack_length=2
    testStack_rc=0 testStack_end=2
pop: ixPop=4 myData=6 myDataPopped=2 testStack_length=1
    testStack_rc=0 testStack_end=1
pop: ixPop=5 myData=6 myDataPopped=1 testStack_length=0
    testStack_rc=0 testStack_end=0
NOTE: Cannot pop empty stack  testStack into myDataPopped
pop: ixPop=6 myData=6 myDataPopped=1 testStack_length=0
    testStack_rc=999999 testStack_end=0
```

```
Stack testStack is empty
```

```
   put;
   %StackDelete(stackName = testStack,
                              rc = testStack_rc
                             );
   put / 'delete: ' testStack_rc=;
run;
```

delete: testStack_rc=0

TEST CODE - QUEUE

```
   data test;

   put /// 'Queue Test' /;

   if _n_=1 then do;
     %QueueDefine(QueueName = testQueue,
                  dataType = n,
                  dataLength = 8,
                  hashexp = 8,
                  rc = testQueue_rc
                     );
     put 'define: ' testQueue_rc=;
   end;

   %QueueLength(QueueName = testQueue,
                QueueLength = testQueue_length,
                rc = testQueue_rc );
   put / 'length: ' testQueue_length = testQueue_rc=;

   %QueueDump(QueueName = testQueue,
              rc = testQueue_rc );

   put;
    do myData = 1 to 5;
   %QueueEnqueue(QueueName = testQueue,
                 InputData = myData,
                 QueueLength = testQueue_length,
                 rc = testQueue_rc
                              );
   put / 'EnQueue: ' myData= myDataEnQueued=
                     testQueue_length= /
         '           ' testQueue_rc= testQueue_new=
                     testQueue_old=;
   end;

   %QueueDump(QueueName = testQueue,
              rc = testQueue_rc );

   put;
   do ixQ = 1 to 6;
   %QueueDequeue(QueueName = testQueue,
                 OutputData = myDataEnQueued,
                 QueueLength = testQueue_length,
                 rc = testQueue_rc
                 );
   put / 'deQueue: ' ixQ= myData= myDataEnQueued =
                     testQueue_length= /
         '           ' testQueue_rc=  testQueue_new=
                     testQueue_old=;
   end;

   %QueueDump(QueueName = testQueue,
              rc = testQueue_rc );
```

Queue Test

Queue testQueue Created. Each Item will take 16 bytes.
define: testQueue_rc=0

length: testQueue_length=0 testQueue_rc=0

Queue testQueue is empty

EnQueue: myData=1 myDataEnQueued=.
testQueue_length=1
testQueue_rc=0 testQueue_new=1 testQueue_old=1

EnQueue: myData=2 myDataEnQueued=.
testQueue_length=2
testQueue_rc=0 testQueue_new=2 testQueue_old=1

EnQueue: myData=3 myDataEnQueued=.
testQueue_length=3
testQueue_rc=0 testQueue_new=3 testQueue_old=1

EnQueue: myData=4 myDataEnQueued=.
testQueue_length=4
testQueue_rc=0 testQueue_new=4 testQueue_old=1

EnQueue: myData=5 myDataEnQueued=.
testQueue_length=5
testQueue_rc=0 testQueue_new=5 testQueue_old=1

Contents of Queue testQueue:
item 1 value 1
item 2 value 2
item 3 value 3
item 4 value 4
item 5 value 5

deQueue: ixQ=1 myData=6 myDataEnQueued=1 testQueue_length=4
testQueue_rc=0 testQueue_new=5 testQueue_old=2

deQueue: ixQ=2 myData=6 myDataEnQueued=2 testQueue_length=3
testQueue_rc=0 testQueue_new=5 testQueue_old=3

deQueue: ixQ=3 myData=6 myDataEnQueued=3 testQueue_length=2
testQueue_rc=0 testQueue_new=5 testQueue_old=4

deQueue: ixQ=4 myData=6 myDataEnQueued=4 testQueue_length=1
testQueue_rc=0 testQueue_new=5 testQueue_old=5

deQueue: ixQ=5 myData=6 myDataEnQueued=5 testQueue_length=0
testQueue_rc=0 testQueue_new=5 testQueue_old=6
NOTE: Cannot Dequeue empty Queue testQueue into myDataEnQueued

Queue testQueue is empty

```
   put;
   %QueueDelete(QueueName = testQueue,
                              rc = testQueue_rc
                             );
   put / 'delete: ' testQueue_rc=;
```

delete: testQueue_rc=0

```
run;
```

EXAMPLE – COMPUTING BETWEENNESS CENTRALITY

The program that follows is an example of using the stack and queue macros to implement Brandes' algorithm for computing the betweenness centrality statistic for a social network. Using the stack and queue macros makes verifying that the code follows the algorithm (shown in the comments) much easier.

```
   /*    see:  Brandes, Ulrik  A Faster Algorithm for Betweenness Centrality */
   /*          Journal of Mathematical Sociology  25(2):163-177, (2001)  */
   data Betweenness(keep = VertexNumber Vertex  CentralityBetween NumberOfNeighbors);
     set &inputDyads end=last nobs=ndyads;
     length VertexID NeighborNumber NeighborID CentralityBetween 8 ;
     length Q_Queue_length Q_Queue_rc 8;
     length w_Neighbor 8;
     length Vertex Stext Vtext Wtext  $ &maxVertexChars;
     format CentralityBetween 10.1;
                                  /*  >> C sub B  <-0 v element of V */
     array CB_Betweenness{&NVertices} 8 _temporary_ (&NVertices*0);
                                  /*  >> P    an array of lists            */
                                  /*  stored as a hyphen separated list of */
                                  /*    vertex IDs                         */
     array P_Via{&NVertices} $ &maxPathLen _temporary_;
                                 /*  << Sigma - number of shortest paths from s to t */
     array Sigma_nShortest{&NVertices} 8 _temporary_ (&NVertices*0);
                                 /*  << d - minimum Path Length from s to t          */
     array d_minimumPathLength{&NVertices} 8 _temporary_ (&NVertices*0);
                                 /*  << delta - dependency of s on v                 */
     array delta_dependency{&NVertices} 8 _temporary_ (&NVertices*0);
                                /*  create a hash listing all the neighbors of a Vertex    */
                                /* NeighborsPerVertex  stores number of neighbors in hash  */
     array NeighborsPerVertex{&NVertices} 8 _temporary_ (&NVertices*0);
                               /*  for each vertex                                 */
     if _n_=1 then do;
       declare hash NeighborsHash(hashexp: 8);
                               /* VertexID - Vertex for which neighbors are stored   */
       rc = NeighborsHash.defineKey("VertexID","NeighborNumber");
                              /* NeighborNumber - ordinal number of neighbor         */
       rc = NeighborsHash.DefineData("NeighborID");         /*  Vertex number of neighbor */
       rc = NeighborsHash.defineDone();
           call missing(VertexID, NeighborNumber, NeighborID);  /*  so SAS won't complain  */
                        /* compute the memory needed for these data                  */
                        /* arrays cb + P + Sigma + d + delta + Neighbors + queue + Stack ) + hash */
       MemoryRequired = &nVertices * ( 8 + &Nvertices + 8 + 8 + 8 + 8 + 16 + 16)
                       + nDyads * 16;
       put 'Memory required ' MemoryRequired  ' bytes';
       put 'Building table of neighbors...';
     end;
                                                    /* load the Neighbors Hash  */
     VertexIDfrom = input(&VertexFrom,VertID.);
     VertexIDto =   input(&VertexTo,VertID.);

     VertexID = VertexIDfrom;
     NeighborsPerVertex{VertexID}+1;                /*  one more neighbor of vertex ID */
     NeighborNumber = NeighborsPerVertex{VertexID}; /*  neighbor number                */
     NeighborID = VertexIDto;                       /*  to is a neighbor of from       */
     neighborAdd_rc=NeighborsHash.add();
```

```
                                                    /*  ---------------------  */
                                                    /*  execute the algorithm  */
                                                    /*  ---------------------  */
  if last then do;
 put 'Analyzing shortest paths...';
    do ixS = 1 to &NVertices;                                  /* >> for s element of V   */
    sText = put(ixS, vertName.);
                                                               /* output a progress meter */
if mod(ixs,&dotEvery) = 0 then put '.' ;
if mod(ixs,&vertexEvery) = 0 then put '----' stext ixs ;
      %StackDefine(stackName = S_Stack,                        /* >> S <=== Empty Stack   */
                  dataType = n,
                  dataLength = 8,
                  hashexp = 8,
                  keyLength = 8,
                  rc = Stack_rc
                  );

      do ixW = 1 to &NVertices;                    /* P[w] <=== empty list w element of V */
        P_Via{ixW} = ' ';
      end;

      do ixT = 1 to &NVertices;     /* sigma[t] <=== 0, t element of V, sigma[s] <=== 1 */
        if ixT ne ixS then Sigma_nShortest{ixT} = 0;
           else Sigma_nShortest{ixT} = 1;
      end;

      do ixT = 1 to &NVertices;            /* d[t] <=== -1, t element of V, d[s] <=== 0 */
        if ixT ne ixS then d_minimumPathLength{ixT} = -1;
           else d_minimumPathLength{ixT} = 0;
      end;
      %QueueDefine(QueueName = Q_Queue,                        /* >> Q <=== Empty Queue   */
                  dataType = n,
                  dataLength = 8,
                  hashexp = 8,
                  keyLength = 8,
                  rc = Q_Queue_rc
                 );


      %QueueEnqueue(QueueName = Q_Queue,                       /*   << enqueue s ===> Q   */
                   InputData = ixS,
                   QueueLength = Q_Queue_length,        /* Note: used by while loop   */
                   rc = Q_Queue_rc
                   );
      do while (Q_Queue_length > 0);                           /*  << while Q not empty do   */
        %QueueDequeue(QueueName = Q_Queue,                     /*   << dequeue v <=== Q  */
                     OutputData = v,
                     QueueLength = Q_Queue_length,      /*  Note: used by while loop  */
                     rc = Q_Queue_rc
                     );
        vText = put(v,vertname.);
        %StackPush(stackName = S_Stack,
                  InputData = v,                                 /*  <<  push v ===> S  */
                  StackLength = S_Stack_length,
                  rc = S_Stack_rc
                  );
        if NeighborsPerVertex{v} > 0 then            /* <<  foreach neighbor w of v do  */
          do ixN = 1 to NeighborsPerVertex{v};
            VertexID = v;
            NeighborNumber = ixN;
                 neighborID = .;
            NeighborFind_rc = NeighborsHash.find();    /* The hash returns NeighborID  */
            w_Neighbor =  NeighborID;
                 wText = put(w_Neighbor,vertName.);
```

```
                                                        /*   <<  if d[w] < 0 then */
            if d_minimumPathLength{w_Neighbor} < 0 then do;
          %QueueEnqueue(QueueName = Q_Queue,             /*   << enqueue w ===> Q   */
                        InputData = w_Neighbor,
                        QueueLength = Q_Queue_length,   /*Note: used by while loop  */
                        rc = Q_Queue_rc
                        );
                                                        /*  <<  d[w] <=== d[v] +1 */
          d_minimumPathLength{w_Neighbor} = d_minimumPathLength{v} + 1;
        end;  /*  d_minimumPathLength{w_Neighbor} < 0  */
                                                        /*  << if d[w] = d[v] +1 */
            if d_minimumPathLength{w_Neighbor} = d_minimumPathLength{v} + 1 then do;
                                          /*  << sigma[w] <=== sigma[w] + sigma[v] */
              Sigma_nShortest{w_Neighbor} = Sigma_nShortest{w_Neighbor} +
                                            Sigma_nShortest{v};
                                                    /*  <<  append v ===> P[w] */
                                                  /* a hyphen separated string  */
          if P_Via{w_Neighbor} = ' ' then P_Via{w_Neighbor} = strip(put(v,8.));
          else P_Via{w_Neighbor} = catx('-',
                                        P_Via{w_Neighbor},
                                        put(v,8.)
                                        );
            end;  /* d_minimumPathLength{w_Neighbor} = d_minimumPathLength{v} + 1 */
        end; /* do ixN = 1 to NeighborsPerVertex{v}  */
  end;    /*  while (Q_Queue_length > 0)                    while Q not empty do  */
  do ixV = 1 to &NVertices;                 /*  <<  delta[v] <=== 0, v element of V  */
    delta_dependency{ixV} = 0;
  end;  /* ixV = 1 to &NVertices */
                                                          /*  while S not empty  */
  do while (S_Stack_length > 0);

    %StackPop(stackName = S_Stack,                        /*  << pop w <=== S  */
              OutputData = w_Neighbor,
              StackLength = S_Stack_length,              /*  used by while loop  */
              rc = S_Stack_rc
             );
     wText = put(w_Neighbor,vertName.);
                                                        /*  << for v element of P[w] */
     ixWN = 1;
     do while (scan(P_Via{w_Neighbor},ixWN,'-') ne ' ');
          v=input(scan(P_Via{w_Neighbor},ixWN,'-'),8.);
       vText = put(v,vertName.);
       ixWN=ixWN+1;
                  /*  << delta[v] <=== delta[v] + sigma[v]/sigma[v]*(1+delta[w])  */
       delta_dependency[v] = delta_dependency(v) +
                             sigma_nShortest(v) / sigma_nShortest(w_Neighbor) *
                                          (1 + delta_dependency(w_Neighbor));
      end;   /*  while scan(P_Via{w_Neighbor},ixWN,'-') ne ' ' */

                                          /*  << CsubB[w] <=== CsubB[w] + delta[w] */
      if w_Neighbor ne ixS then do;
        CB_Betweenness{w_Neighbor} = CB_Betweenness{w_Neighbor} +
                                     delta_dependency(w_Neighbor);
        end;   /*  w_Neighbor ne ixS */
  end;  /* while S_Stack_length > 0  */
                                                /*  ends loop over s element of V - */
                                                  /*clear out stack and queue  */
  %StackDelete(stackName = S_Stack,
               rc = Stack_rc  );
  %QueueDelete(QueueName = Q_Queue,
               rc = Queue1_rc );
end;   /* ixS=1 to &NVertices for s element of V */
```

```
                                                    /*    ---------------------------------  */
                                                    /*    output the results to the dataset  */
                                                    /*    ---------------------------------  */
        do vertexNumber = 1 to &NVertices;
          vertex = put(vertexNumber,vertName.);
          CentralityBetween = CB_Betweenness{vertexNumber} / 2;
          NumberOfNeighbors = NeighborsPerVertex{vertexNumber};
          sigma = sigma_nShortest(vertexNumber);
          delta = delta_dependency(vertexNumber);
          output;
        end;  /* ixV = 1 to &NVertices */
    end;  /* if last    execute the algorithm  */
run;
```

REFERENCES

- Brandes, Ulrik *A Faster Algorithm for Betweenness Centrality* Journal of Mathematical Sociology 25(2):163-177, (2001)
- Hoyle, Larry *Visualizing Two Social Networks Across Time with SAS®: Collaborators on a Research Grant vs. Those Posting on SAS-L.* SAS Global Forum 2009 paper 229-2009, Washington D.C., 2009
- Knuth, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xx+650pp. ISBN 0-201-89683-4

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Sample code associated with this paper can be found at:

http://www.ipsr.ku.edu/ksdata/sashttp/SGF2009/

Contact the author at:

Larry Hoyle

Institute for Policy & Social Research, University of Kansas

1521 Lilac Lane, Suite 607 Blake Hall

Lawrence, KS, 66044-3177
785-864-9110

LarryHoyle@ku.edu

www.ipsr.ku.edu