

Flexibility of SCL Lists

Scott Groussman, Value Health Management, Dallas, TX

Robin Yeldell, Value Health Management, Dallas, TX

ABSTRACT

SCL Lists, a new data structure implemented in SAS, version 6.07, provide many new programming techniques to SCL programmers. Through examination of a classic problem, the following list techniques and flexibility will be discussed:

- Array Alternative - SCL lists can effectively replace single and multi-dimensional arrays
- Dynamic Structure - List items can be dynamically added, deleted, and rearranged
- Direct Access - List items can be accessed by an index value or by a name
- Argument Passing - Lists can greatly simplify the passing of a large number of arguments
- Data Set Alternative - Lists make convenient tables and can be accessed faster than data sets
- Data Sharing - List items can be stored in an environment list and made common to other routines
- Storage Option - Lists can be saved to disk for later retrieval

INTRODUCTION

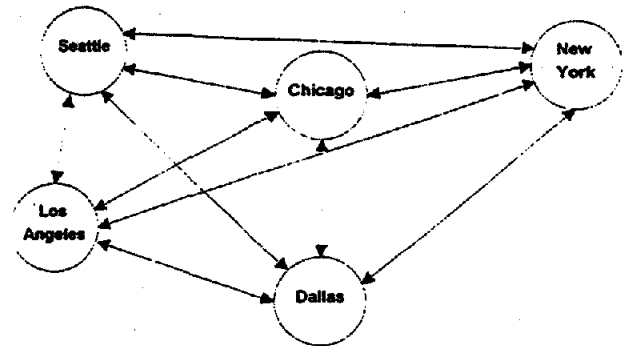
SCL Lists, a new data structure implemented in SAS, version 6.07, provide many new programming techniques to SCL programmers. While lists are ordered collections of data similar in many ways to arrays, lists provide virtually unlimited run-time functionality. Through the classic Traveling Salesperson problem, the techniques and advantages of lists over traditional data types will be explored.

THE TRAVELING SALESPERSON PROBLEM

Suppose a salesperson has been assigned a region covering an arbitrary number of cities. The salesperson needs to travel to each of these cities for business meetings. Each city has roads connecting it to every other city. The salesperson would ideally travel the shortest route to make the most efficient use of his resources (he is working off commission). Therefore, what is the shortest route that visits each city once, and only once, and ends at the originating city?

For example, assume the salesperson needs to visit Seattle, Dallas, Chicago, New York, and Los Angeles and wants to travel the shortest possible route and return to the originating city (Figure 1).

Figure 1 Graphical Representation



The Traveling Salesperson problem is related to many common computer science problems. In general, the Traveling Salesperson problem contains objects and interconnections between these objects. Electronic circuits, telecommunications linking, network flow, scheduling, inventory, and many other problems have these similar properties. Our salesperson is concerned with distance, but the object interconnections could easily be weighed by factors such as time, money, and capacity. Furthermore, the interconnections can be directional: a path from object A to object B does not imply a corresponding path from Object B to Object A.

The first step in writing a program to solve this problem determining how we are going to represent this data in a computer. Three possibilities come to mind:

- Table Data Set
- Two-Dimensional Array
- SCL Lists

1. Table Data Set

Each interconnection, or road, could be represented as observations in a SAS data set as described in Table 1.

Table 1 Data Set Representation

From	To	Distance
Seattle	Dallas	2131
Seattle	Chicago	2052
Seattle	New York	2841
Seattle	Los Angeles	1134
Dallas	Seattle	2131
Dallas	Chicago	921
Dallas	New York	1559
Dallas	Los Angeles	1399
Chicago	Seattle	2052
Chicago	Dallas	921
Chicago	New York	809
Chicago	Los Angeles	2048
New York	Seattle	2841
New York	Dallas	1559
New York	Chicago	809
New York	Los Angeles	2794
Los Angeles	Seattle	1134
Los Angeles	Dallas	1399
Los Angeles	Chicago	2048
Los Angeles	New York	2794

However, determining the shortest route will require accessing this data set many times. Because disk access is much slower than internal memory access, the solution will undoubtedly require accessing the observations in this data set randomly, which is even slower than sequential access.

2. Two-Dimensional Array

A two-dimensional array is a convenient storage mechanism and can be stored in memory, thereby avoiding costly disk access. The array would have the "From" city in the first dimension index and the "To" city in the second dimension index, with the element value being the distance.

Table 2 illustrates how this array might look in memory.

Table 2 Two Dimensional Array Representation

	Seattle	Dallas	Chicago	New York	Los Angeles
Seattle		2131	2052	2841	1134
Dallas	2131		921	1559	1399
Chicago	2052	921		809	2048
New York	2841	1559	809		2794
Los Angeles	1134	1399	2048	2794	

Array referencing is immensely faster than disk access and random access is straight forward: simply supply the two indices. However, arrays impose some inherent disadvantages: array sizes must be declared at program compilation and cannot be re-sized during program execution. The dynamic programmer is confronted with a dilemma: either right-size the array and restrict the programs functionality in favor of memory conservation, or oversize the array and waste memory in favor of program flexibility. Additionally, arrays commonly have unused cells yet are another waste of memory.

In the Traveling Salesperson problem, diagonal cells are not used. This problem becomes more pronounced if every city is not reachable from every other city; fewer interconnections means more unused cells.

3. SCL Lists

Lists can be thought of as a dynamic array although you can define lists with a fixed number of elements. These elements can be directly referenced much in the same way an array is directly referenced. However, lists have many capabilities not available with arrays. The most notable advantage is dynamic sizing, the capability to control the number of data elements to be stored. New items can be inserted into the list at any point in a program, and the list will automatically re-size itself. Likewise, old items can be deleted. Consider using a list if you ever ask yourself: "How large should I define this array?"

Why introduce a limit to the number of elements when you could use a list, which is capable of expanding itself as needed?

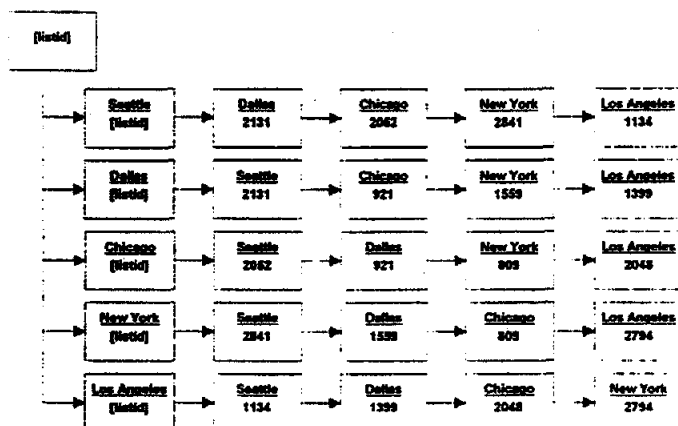
When defining an array, the element data type must be declared as either numeric or character. Lists do not have these restrictions, and list items act independently of one another. Lists can contain any combination of numbers, characters, or even other lists, which makes them a useful storage mechanism for disparate data types.

While array elements can only be referenced by the index number in the array, list items can be referenced a variety of ways. Items can be referenced by supplying the index number in the list, as with arrays. Additionally, list items can be named and then directly accessed by these names.

Lists can be written to disk and loaded in future sessions, a significant advantage over arrays and macro variables, which are not written to disk. This capability, coupled with the dynamic flexibility of lists, virtually eliminates the need for parameter data sets. Extended tables driven from lists are faster than those driven from data sets for the same reasons memory addressing is faster than disk access.

Figure 2 demonstrates how we could use a list to store the data required for the Traveling Salesperson.

Figure 2 List Representation



The base list contains a sublist for each city. These sublists are conveniently named for the city they represent. Each city's sublist contains a number of named numeric items. Each sublist item

represents the distance between the base list city and the city indicated by the item's name. For instance, the first item in Seattle's sublist is named Dallas and has a value of 2,131. Therefore, this item indicates that the distance between Seattle and Dallas is 2,131 miles.

The following method could be used to create such a list structure:

```

LENGTH FROM TO $11
      DISTANCE 8;
LOADLIST: METHOD INDSN    $17
      SAVEDSN  $25
      CITYLIST 8;

DSID = OPEN(DSN, 'I');
CITYLIST = MAKELIST();
IF DSID AND CITYLIST THEN DO;
  CALL SET(DSID);
  DO WHILE (^FETCH(DSID));
    IF ^NAMEDITEM(CITYLIST, FROM) THEN DO;
      ROADLIST = MAKELIST();
      CITYLIST = INSERTL(CITYLIST,
        ROADLIST, -1, FROM);
    END;
    ELSE ROADLIST = GETNITEML(CITYLIST,
      FROM);

    ROADLIST = INSERTN(ROADLIST, DISTANCE,
      -1, TO);
  END;
  RC = CLOSE(DSID);
END;
ELSE PUT 'ERROR: FILE COULD NOT BE OPENED';

RC = SAVELIST('CATALOG', SAVEDSN, CITYLIST);
IF RC THEN PUT 'ERROR: LIST NOT SAVED';
ENDMETHOD;

```

This method creates a new, empty list and then builds the list from a parameter data set (as defined earlier). Notice that the number of cities is not predetermined. As a new city is encountered in the data set, a new sublist is created. Furthermore, each city's sublist only contains items for the interconnections found in the data set. Memory is more effectively utilized as compared with unused array elements. Because we might need this list more than once, the list is saved, allowing us to avoid rebuilding the list more than once. Now that we have selected the use of SCL lists as our method of accessing the data, possible solutions can be considered.

BRUTE FORCE SOLUTION

Those with access to a mighty mainframe, no chargeback, and limitless CPU resources might answer the salesperson's problem with the following: for every possible combination, determine the distance and take the shortest path. Determining the possible permutations when every city is reachable from any city is fairly simple: do nested for loops, checking every possible combination, and retain the shortest path encountered (care must be taken, however, to ensure each city is visited only once). This method is called brute force because each possible scenario is checked.

With only a few cities, brute force is acceptable because a manageable number of combinations exist. However, the number of combinations quickly increases as more cities are considered.

Table 3 shows the number of possible routes available to our salesperson.

Table 3 Factorial Growth Function

Number of cities	Number of Permutations
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800

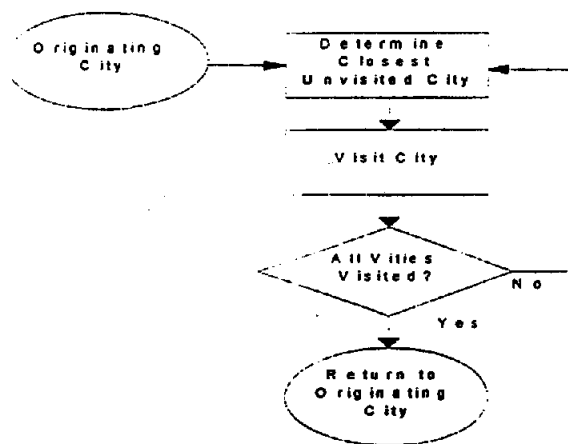
The number of combinations is X factorial (where X is the number of cities). Brute force quickly becomes unfeasible; however, it is the only way to absolutely know the shortest path.

APPROXIMATION

Is the absolute shortest path necessary? What about finding "one of the shortest" paths? For this example and many others, an approximation is satisfactory. Therefore, let's just find a reasonably short path. To do this, the salesperson might just start in a city and travel to the closest city not yet visited. If this were done at each city visited, then a reasonably short path can be found without determining all the possible paths.

Figure 3 shows the flow of such an algorithm.

Figure 3 Approximation Program Flow



Determining the next closest city is simple if we take advantage of SCL's SORTLIST feature. Each city's list of distances to neighboring cities can be sorted in ascending order so that the closest city appears first in the list. Adding the following code to our LOADLIST method prior to saving the list would achieve this effect.

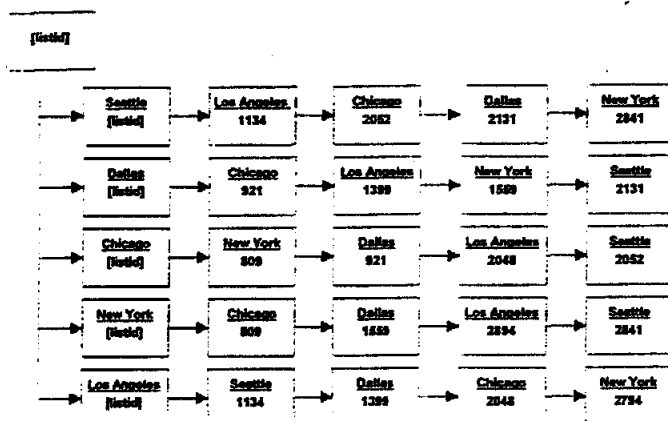
```

DO I = 1 TO LISTLEN(CITYLIST);
  CITYLIST = GETITEML(CITYLIST, I);
  CITYLIST = SORTLIST(CITYLIST,
    'ASCENDING VALUE');
END;

```

The new list structure would then appear as in Figure 4:

Figure 4 Sorted List Representation

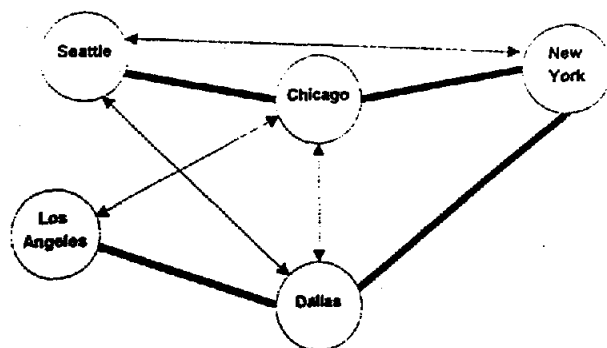


FORK IN THE ROAD

At this point, we have assumed that each city is reachable by every other city. What happens if we assume an airline analogy where you are at the mercy of the airlines and from a given city, you can only reach certain other cities? Finding possible routes becomes more difficult when dead ends occur from a given city from which no other unvisited cities are directly accessible. Therefore, we have to backtrack when a dead end is encountered.

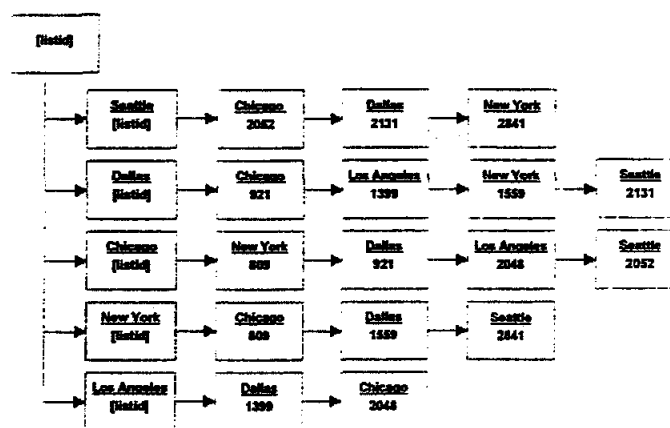
Figure 5 demonstrates how a path starting in Seattle reaches a dead end by the time we reach Los Angeles. Without a connection between Los Angeles and Seattle, the salesperson would not be able to return home by air.

Figure 5 Dead End



We can see what affect this has on the list structure in Figure 6.

Figure 6 Sorted List Representation with Dead Ends



Backtracking will require tracking the cities already visited in such a way that the path can be undone for one or more cities and then started back up. In Figure 4, the salesperson would need to back up to Chicago, visit Los Angeles, Dallas, New York, and then return to Seattle. A Depth First Search algorithm, widely used in solving many problems as the Traveling Salesperson, can be employed to provide this functionality.

```
DFS:METHOD CITYLIST 8;
  PATH = MAKELIST();
  DONE = 'N';
  DO I = 1 TO LISTLEN(CITYLIST);
    IF ^SEARCHC(PATH,NAMEITEM(CITYLIST,I))
      THEN CALL
        METHOD('TRAVSALE.TRVSALE.METHODS.SCL',
          'VISIT',CITYLIST,PATH,
          NAMEITEM(CITYLIST,I),DONE);
    IF DONE = 'Y' THEN LEAVE;
  END;
  RC = DELLIST(PATH,'Y');
ENDMETHOD;

VISIT:METHOD CITYLIST PATH 8 VISIT $11 DONE $1;
  PATH = INSERTC(PATH,VISIT,-1);
  ROADLIST = GETNITEML(CITYLIST,VISIT);
  IF LISTLEN(PATH) = LISTLEN(CITYLIST)
    THEN DO;
      IF NAMEDITEM(ROADLIST,GETITEMC(PATH,1))
        THEN DO;
          DONE = 'Y';
          CALL PUTLIST(PATH,'PATH FOUND : ',3);
        END;
      END;
    ELSE DO I = 1 TO LISTLEN(ROADLIST);
      IF ^SEARCHC(PATH,NAMEITEM(ROADLIST,I))
        THEN CALL
          METHOD('TRAVSALE.TRVSALE.METHODS.SCL',
            'VISIT',CITYLIST,PATH,
            NAMEITEM(ROADLIST,I),DONE);
      IF DONE = 'Y' THEN LEAVE;
    END;
  RC = DELITEM(PATH,SEARCHC(PATH,VISIT));
ENDMETHOD;
```

The DFS method first creates an empty list call PATH, which will be used to store the cities visited. The DONE flag is initialized and then a loop through each city is entered. VISIT is called on the first city in the city list. VISIT immediately marks the city as

visited. A check is made to see if all cities have been visited, and if the original city is reachable, the path is printed and the DONE flag is set. Otherwise, a recursive call to VISIT is made to each unvisited city. If the loop terminates normally and not every city has been visited, then we have reached a dead end and the city is removed from the path.

In Figure 4, this program originates in Seattle. The closest unvisited city to Seattle is Dallas and so our salesperson travels to Dallas. From Dallas he travels to Chicago, and then to New York. Once he reaches New York, he hits a dead end. The only unvisited city is Los Angeles, which is not reachable from New York. Therefore, the algorithm backtracks to Chicago and tries Los Angeles. This is just as much a dead end as before: he cannot reach New York, the last unvisited city. The algorithm continues in this fashion until the first complete path is found: Seattle, Dallas, Los Angeles, Chicago, New York.

Another advantage of lists over arrays are clearly visible from this example. The recursively called VISIT method is passed the city list. While arrays can be passed as parameters in SCL, the behind-the-scenes actions often result in undesirable effects. When SCL passes an array, a reference to each element is passed to the program being called. When a list is passed, only the base list identifier is passed. All other element references are accessed via the based list pointer. Thus, passing large arrays significantly increases the processing time for function calls.

Another consideration for parameter passing is recursion. This example will have multiple nested calls to itself. If arrays were used, each nested call would require its own set of references. As the number of active nested calls increases, memory for all the array references soon becomes scarce. Determining a route with even 50 cities becomes problematic.

An improvement in our algorithm would be to store the path and city lists in the environment list. SCL provides for a global and local environment list. The global list can be accessed by all SAS/AF® applications within the current SAS session while the local list can be accessed by any SCL program within the SAS/AF application. These environment lists make convenient storage spaces for commonly used variables and tables, such as our city list. Parameter passing for these items can be completely avoided.

CONCLUSION

The problem of the traveling salesman, as well as any problem involving a dynamic data complex, is greatly simplified by the dynamic nature of list structures. Lists offer many advantages over arrays and disk storage for rapid access and irregular data patterns. Because lists are available to all child processes by passing a single 8-byte pointer saves considerable framing overhead when a significant amount of data has to be passed between routines.

Lists, when combined with other recent developments in SCL, can revolutionize the way you solve problems in all of your SCL programming efforts. Many of the techniques that are facilitated by lists also facilitate more flexible, reusable program modules. Probably the worst drawback to lists is getting accustomed to using this data structure in your SCL programming, when you find yourself trying to code something in a data step without a comparable structure.

One note of caution: lists are stored in memory, creating the chance that you will make inappropriate use of memory when your lists get very large. Most operating systems will swap memory storage out to disk when you overrun your "chip" memory, which is a costly imposition. Therefore, lists are ideal

only if the expected amount of data will be handled without swapping memory out to disk.

ABOUT THE AUTHORS

Scott Groussman and Robin Yeldell used lists extensively when designing and implementing a proprietary health care information delivery and reporting system. This system, presently in use by many health care carriers, purchasing alliances, and private companies around the nation, provides analytical support to better manage medical costs for over one million people.

They began system development as a Beta Test Site for SAS release 6.08 for Windows in November 1992 and have used frames and methods exclusively in developing the extensive user interface. The system is now supported on the Windows®, OS/2, and Unix platforms.

SAS and SAS/AF are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. OS/2 is a registered trademark of International Business Machines. Windows is a registered trademark of Microsoft Inc.

® indicates USA registration.