DATA STRUCTURES FOR PATTERN RECOGNITION ALGORITHMS: A Case Study

SLAC-PUB-1566
March 1975

Charles T. Zahn, Jr.
Computation Research Group
Stanford Linear Accelerator Center*
Stanford, California 94305

Summary

This paper will describe experiences gained while programming several pattern recognition algorithms in the languages ALGOL, FORTRAN, PL/1 and PASCAL. The algorithms discussed are for boundary encodings of two-dimensional binary pictures, calculating and exploring the minimum spanning tree for a set of points, recognizing dotted curves from a set of planar points and performing a template matching in the presence of severe noise distortions. The lesson seems to be that pattern recognition algorithms require a range of data structuring capabilities for their implementation, in particular arrays, graphs and lists. The languages PL/1 and PASCAL have facilities to accommodate graphs and lists but there are important differences for the programmer. The ease with which the template matching program was written, debugged and modified during a 3 week period, using PASCAL, suggests that this small but powerful language should not be overlooked by those researchers who need a quick, reliable, and efficient implementation of a pattern recognition algorithm requiring graphs, lists and arrays.

Algorithms

Encoding Digital Pictures

The storage of digital pictures[1] generally requires a rectangular array of picture elements (pixels), each represented by a small integer. The large number of pixels and the small number of bits (1 to 8) per pixel suggest packing the array with one machine word containing several pixels. For efficiency in performing local preprocessing on pixel neighborhoods, it seems natural to unpack several rows of the digital picture and then repack.

Binary digital pictures consist of connected regions of black or white color which can be represented by a set of polygonal boundary curves. One method[2] for calculating these curves scans the binary picture from top to bottom, extracting curvature points where some boundary curve changes direction. These curvature points are maintained in several linked lists which grow and merge and eventually become cyclic lists corresponding to a completed closed polygon (see Figure 1). There is also a natural insidedness relation among these non-intersecting boundary curves which can best be represented as a directed rooted tree of curves.

A similar method for binary pictures on a triangular grid[3] requires a top-down processing of the strips (corridors) between two adjacent picture rows (see Figure 2). The small pieces of boundary which touch the edges of the corridor are recognized as tops or bottoms according to which edge they touch, and by maintaining a queue of bottom elements, the processing of corridor sequences becomes quite simple: bottoms are added to the queue and tops are linked to the first queue element which is then taken off the queue. This queueing procedure doesn't even need to know where one corridor leaves off and the next begins.

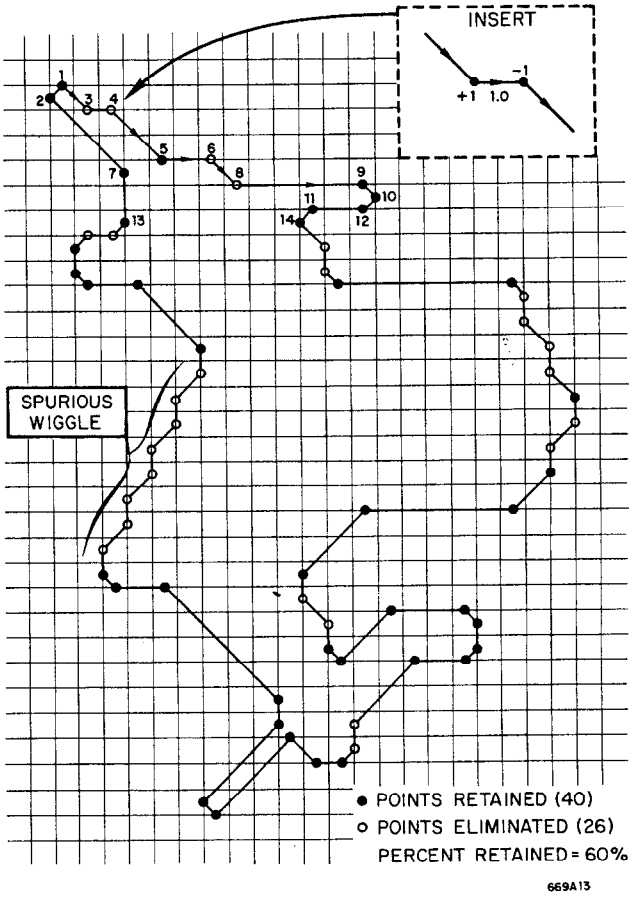

FIGURE 1. Region boundary

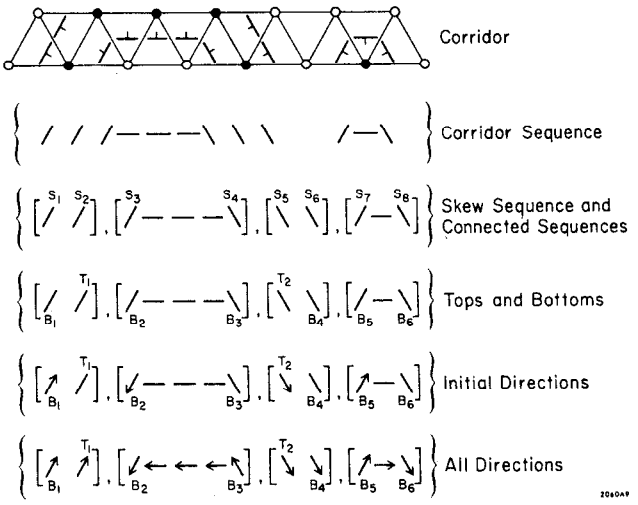

FIGURE 2. Corridor sequence

*Work supported by United States Energy Research and Development Administration.

(To be presented at Computer Graphics, Pattern Recognition, and Data Structure Conference, Los Angeles, California, May 14-16, 1975)

Processing Region Boundaries

It is often appropriate to smooth the polygonal region boundaries by the elimination of wiggly sequences of curvature points caused by quantization noise (see Figure 1). This requirement provides further motivation for representing the curves as linked lists. Not only is the deletion more convenient, but the storage for the deleted points can be re-used for other curves. Computing the convex hull of a closed polygonal curve retains more information if the portion of boundary delineating each concavity is extracted to form a polygonal boundary for the concavity. Once again the flexibility of linked lists is useful.

For the computation of area and local curvature[4] as well as Fourier Descriptors of the boundary shape,[5] it is very convenient to have the boundary represented as a cyclic doubly linked list in close correspondence to the true geometric relationship of the curvature points. The storage for Fourier Descriptors is naturally a static array.

Cluster Analysis

An important problem in pattern recognition is the description of structural properties of a set of points in a multidimensional space. One approach to this problem which applies in a general metric space[6] is to compute the minimal spanning tree (MST) from the complete graph of points with metric distance as edge-weight (see Figure 3). Each point can be represented as an array of coordinate values, and the set as an array of points. The most efficient algorithm for constructing the MST begins with a single node (point), and repetitively adds a new node and edge to the growing tree until all points are nodes of the tree. Since the tree grows and the local degree (number of connecting edges) of nodes is not known a priori, it is convenient to associate with each node a list of node-edge adjacencies which refer to actual edges. Each edge has length information and is referred to by two node-edge adjacencies. Two arrays are used to select the next edge to add to the growing tree.

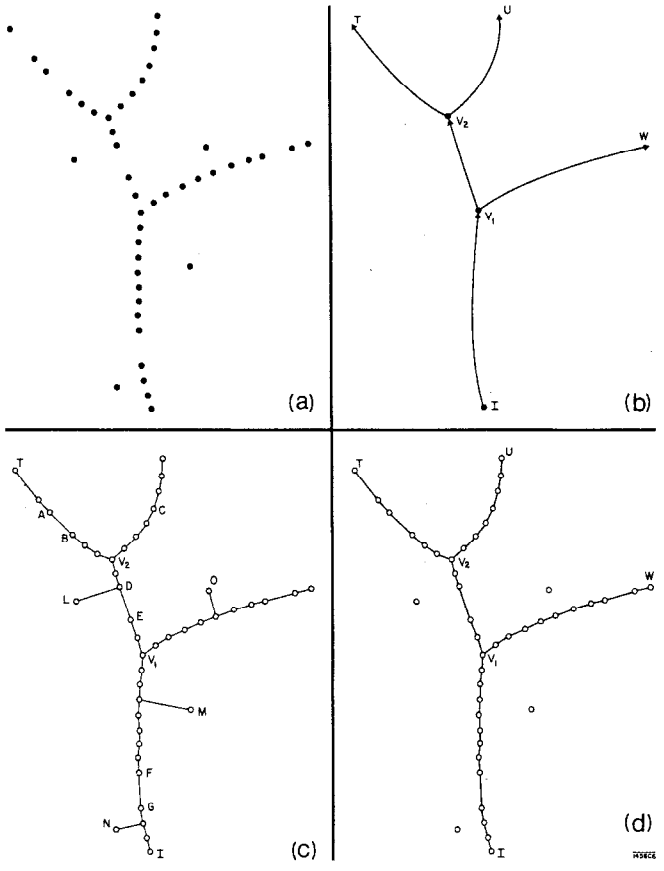

FIGURE 3. Particle tracks problem. (a) Artificial particle track photo. (b) Tracks and interaction vertices. (c)Minimum spanning tree for (a) showing "hairs." (d) MST without "hairs."

Well-defined clusters can be found by deleting certain "inconsistent" edges of the MST; to determine inconsistency, one must compute simple statistical properties of small sets of edges near the two end-nodes of a given edge. Forming a list of the edges of a subtree can be done by a recursive procedure or using an explicit stack. The connected subtrees which result from deletion of inconsistent edges can be further investigated by calculating the diameter path (longest), considering it as a one-dimensional domain and plotting other functions (e.g., point density estimates) along this coordinate. An especially elegant computation can be constructed to determine, for a given node-edge adjacency, the maximum path length between the given node and all nodes of the subtree defined by the given node-edge adjacency. These "relative-depths" allow almost immediate calculation of "relative-diameters" and the "global" diameter. They are properties of the node-edge adjacencies and would have been awkward to include in the data-structure if lists of node-adjacencies had not been directly represented. When the diameter path has been found, it may be convenient to represent it as a list or array, depending on the subsequent uses of the path.

Dotted Curve Recognition

As implied by Figure 3, the MST can be useful for the recognition of curves formed by a set of points in the plane. To realistically apply the MST to recognize particle tracks from physics photographs,[7] some shortcuts were necessary. The scanner transforms the photo into a list of points corresponding to the normal top-down left to right TV raster scan of a rectangular area. The sorted order of rows was exploited for its geometric content in the following way: a quasi-minimal spanning forest (Q-MSF) was constructed by restricting the tree to edges connecting a point to other points in a fixed rectangular window around the point. Because of this modification, it became appropriate to maintain a queue of current candidate edges for entry into the tree sorted on edge length (i.e., a priority queue[8]). The crucial aspect of this method is that searching the rectangular window around a new MST node requires looking at only a small horizontal strip of the entire picture -- a small subset of the sorted rows. With these modifications, it became feasible to compute quasi-minimal spanning forests for 1000-2000 points. To economize on storage requirements, the Q-MSF was represented as directed rooted trees with each new edge pointing back into the growing tree. Although this restricts the subsequent tree explorations to follow directed paths, for this type of line-like data the restriction was quite tolerable.

The procedure for curve recognition extracts directed paths from the Q-MSF (possibly after deletion of "hairs": see Figure 3), and applies a variant of the iterative endpoint fit method[9] to recursively decompose the path into approximately linear segments.[7] This requires recursion or an explicit stack. Figure 4 depicts how the method works by breaking path (AB) at C, accepting (AC) as a sufficiently linear segment, breaking (CB)

at D, and then accepting (CD) and (DB). The resulting segments are connected into lists which represent curves with slowly varying direction. It is useful to provide links which associate each segment back down to the lower level path of points constituting it.

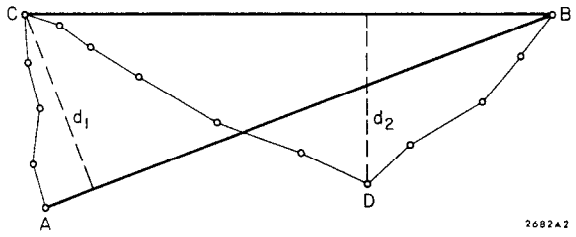

FIGURE 4. Iterative endpoint fit

Noisy Template Matching

A program has been designed[10] to recognize a partial fragment of an original base point set in the plane even after rotation, uniform scale change, and extremely heavy noise distortions. The method used depends on invariance of local structure of the MST to these forms of distortion. Figure 5 shows the MSTs for

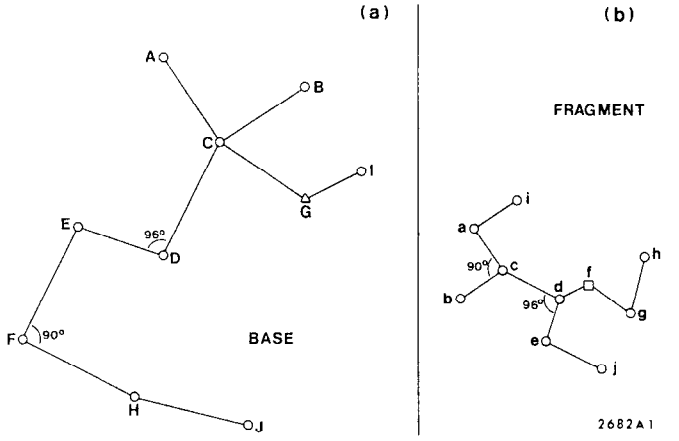

FIGURE 5. Local structure of MSTs

two almost matching sets; the local structure invariant to scale changes consists of the angles formed by the sequence of edges about a node, as well as the ratios between lengths of pairs of edges subtending such angles. To adjust to these needs, we calculate a direction for each node-edge adjacency while growing the MST, and keep each node's adjacency list ordered by edge-direction counter-clockwise around the node. When the MST is complete, we calculate angles around each node, storing the information at node-edge adjacencies. The minimum angle at each node is calculated along with the ratio between the lengths of the edges subtending this minimum angle. This information is stored at the node. The nodes of the MST are next arranged into separate lists based on local degree (≤ 6 for a planar MST), and each list is ordered by the value of minimum angle. This structure is computed for both base and fragment point sets so that the matching algorithm will be very efficient. Attempts to find nodes with similar local structure restrict their attention to nodes of nearly the same degree and with compatible minimum angle and length ratio. This focuses the search dramatically.

When there is reasonable evidence of matching local structure between two nodes, then a global least squares fit is performed to obtain a final verification. This requires a 4x4 matrix which defines the linear system to be solved.

Data-Structures

The functional requirements of the various algorithms described above dictate the use of a wide range of data structures including arrays, matrices, singly and doubly linked lists, cyclic lists, rooted directed and symmetric trees, simple and priority queues, stacks, and sorted lists. These are fundamental data-structures for general programming,[11] and pattern recognition methods appear to need a rich blending of the entire range for their convenient implementation. We have attempted to describe the algorithms in sufficient detail to indicate that the chosen data-structures were the natural consequence of certain requirements for efficiency or convenience.

Lists, trees, queues, and stacks can be implemented in terms of two more primitive data-types called records and references. A record is a data-structure consisting of several named fields of possibly different types, and a reference is a variable which points to some record. If records containing references to other records can be dynamically created and destroyed during the execution of a program, then one has all the facilities needed to create arbitrary graphs; lists, trees, queues and stacks are special instances of general graphs.

While there are ways to implement lists and trees in some cases using arrays, the lack of dynamic storage allocation can make such solutions awkward. The clarity of programs can also suffer when arrays are used for purposes never intended.

Language Comparison

FORTRAN and ALGOL-60

From the point-of-view of data-structures, FORTRAN and ALGOL-60 are almost identical since arrays are all that is offered. The curvature points algorithm[2] was implemented in ALGOL-60 (see Appendix A,B in[12]) with arrays named X,Y,EDGEIN,EDGEOUT to represent the curvature points themselves, and arrays named TOP, BOTTOM, NEXTPOINT, NEXTEDGE, LASTPOINT, LASTEDGE, ANGLE, LENGTH to represent the cyclic polygonal curves. The program is reasonably clear but there is some waste of storage, and misuse of the integer pointers cannot be detected by the language compiler as it can be in languages with references declared as bound to one particular record class.

The dotted curve recognition program for particle tracks[7] was implemented in FORTRAN although originally developed and debugged in PL/1. The clarity of the program suffered considerably in the translation while the efficiency was not substantially enhanced. It was probably quite fortunate that the program was correct before being translated to FORTRAN since the array implementation of pointers has the same problem as mentioned above for ALGOL-60.

The MST cluster analysis algorithm has been programmed in FORTRAN[13] making it more accessible, but the FORTRAN version lacks something in clarity, for the usual reasons, even though the author made a valiant attempt.

PL/1

The PL/1 language offers structures (similar to records) and pointers (references), and allows for dynamic storage allocation and deallocation. Unfortunately, pointers are not restricted to refer to a particular class of structure and, as a result, detecting misuses of pointers by traditional debugging can be quite pain-

ful. We implemented the minimal spanning tree clustering[6] in PL/1 as well as the earlier versions of the dotted curve recognition.[7] In general, we found PL/1 to be an adequate programming tool for these algorithms if used in a very constrained way, avoiding the more dangerous or mysterious aspects of the language.

ALGOL-W

The language ALGOL-W[14] (implemented extremely well on the IBM/360) is based on ALGOL-60 but includes records separated into disjoint classes, and references which are restricted to refer to a particular class or set of classes. The compiler can, therefore, diagnose most misuses of references and save the programmer many grey hairs. The algorithms we have implemented in PL/1 could have been programmed almost identically in ALGOL-W with a gain in efficiency as well as programmer convenience.

PASCAL

The language PASCAL[15,16,17] is based on ALGOL-60 and ALGOL-W, but implements many of the important data structuring facilities described and motivated by Hoare.[18] It has the records and references (here called pointers) of ALGOL-W but includes programmer defined data types, constants, sets, etc. It enjoys most of the desirable properties of ALGOL-W but was more consciously designed to support modern ideas of hierarchical refinement of data-structures and procedures.

We programmed, debugged, experimented with, and modified the noisy template matching algorithm[10] as well as writing the paper -- all within 3 weeks. The initial computer run was an attempted compilation of ~900 lines of PASCAL and the final production run occurred 8 working days later with a 1300 line program! Only two programming errors slipped past the compiler and they caused no severe difficulty. This was, moreover, our first serious effort at programming in PASCAL, although familiarity with ALGOL-W helped.

In PASCAL, the programmer can define the type pixel to be a subrange 0..7 of integer values, and then a packed array [1..500,1..500] of pixel would require only 25,000 32-bit words of storage. The programmer must consciously pack and unpack, but need not be concerned with the machine dependent details of shifting, masking and field extraction.

SGOL75

SGOL75 is a language which is currently under experimental development, and which represents an attempt to implement some of the features of PASCAL, along with structured control as advocated by Knuth,[19] using the macro-translator for MORTRAN2.[20] The language MORTRAN2 is a structured extension to FORTRAN which can be translated into standard FORTRAN by a very small standard FORTRAN program (700 cards). The translator is driven by a list of macro-rules for text replacement, and SGOL75 → FORTRAN translation is achieved simply by using a list of macro-rules appropriate to SGOL75. The most interesting development to date has been the relative ease with which records and references can be implemented[21] with very good protection against misuse of references.

The macro-based implementation of a structured language allowing records and references by a standard FORTRAN program which translates the given language into standard FORTRAN has large implications. The major advantages of FORTRAN (i.e., frequency of implementation and program libraries) are retained while its considerable deficiency as an intellectual tool for problem solving is not relevant. Other data structures such as stacks and queues are easy to implement in the macro-based fashion, as are other user-defined facilities. In its present state of development, SGOL75 would be a fairly convenient vehicle for programming the pattern recognition algorithms discussed above.

PASCAL Data-structures

The PASCAL language has data-definition and data-structuring facilities which, in conjunction with records and references, allow the programmer to structure data in a conceptually natural way. The programmer may define a name (i.e., identifier) to be synonymous with a constant value (e.g., PI=3.14159). He may define a new type (i.e., range of values) as a finite set of distinct names which become the constant values of the new type (e.g., Color=[Red,Yellow,Green]) or as a subrange of the integers (e.g., Age_range=0..150).

We shall present some concrete examples of PASCAL data-structures in the context of pattern recognition algorithms.

General Digital Pictures

Suppose we wish to represent general digital pictures -- that is, colored (trispectral) as well as black and white. We begin by defining several constant names which will be used uniformly in all subsequent data-definitions, data-declarations and commands.

```
constant Pix_size = 100; Black=63; White=0;
```

Next, we define four new types - Color, Pixel, Pix_type and Pix_range

```
type   Color     = [Red,Yellow,Green];
       Pixel     = White .. Black;
       Pix_type  = [Colored,Black_white];
       Pix_range = 1 .. Pix_size;
```

and then new structured types called Simple_pix and Colored_pix

```
type  Simple_pix = packed array[Pix_range,Pix-range]
                   of Pixel;
      Colored_pix = array[Color] of Simple_pix;
```

Now we can define the new type Picture

```
type  Picture =
       record
          Name: Text_string;
          case P_T: Pix_type of
            Colored: (C_P : Colored_pix);
            Black_white: (S_P : Simple_pix)
       end;
```

The case variant construction within a record is peculiar to PASCAL; in this case, it means that each variable of type Picture will have a field called P_T of type Pix_type (i.e., Colored or Black-white) and subsequent fields will have names and types depending on the particular value of P_T. We have thus defined a single data-type which can be used to represent general digital pictures.

Should we need to handle pictures 150x150 with picture elements in the range 0=White to 15=Black, then all that is required is to change the constant definitions for Pix_size and Black! All other adjustments required throughout the program are obtained consistently by recompilation of the program.

Planar MST for Matching

The PASCAL data-structures employed in the template

matching problem[10] included the following three new types to represent the MSTs and their associated angle and length information. The symbol (↑) means "reference to".

```
type  Point_type = array[1..Max_dim] of real;
      Node_type  =
         record
             X:Point_type;Degree:Integer;
               Next:↑Node-type;
             Min_angle,Length_ratio:real;
             First_adj,Min_adj:↑Adj_type
         end;
      Adj_type   =
         record
             Edge:↑Edge_type;Next:↑Adj_type;
             Direction,Angle:real
         end;
      Edge_type  =
         record
             End_node:array[1..2]of↑Node_type;
             Length:real
         end
```

Inside the procedure which actually grows the MST, we have the following variable declaration

```
variable Near : array [1..NN_max] of
     record Node:↑Node_type;Distance:real;
                  Free:boolean end;
```

Each node/point has an integer index between 1 and NN_max and Near[i].Node references that node of the current tree nearest to the ith node if Near[i].Free is true and Near[i].Distance is the distance between these two nodes. Free means "not yet in the tree".

Node lists based on local degree (≤6) and sorted on Min_angle require

```
type Node_lists = array [1..6] of↑Node_type;
variable Base_lists, Fragment_lists:Node_lists;
```

These examples do not involve all the nice data-representation facilities of PASCAL, but we hope to have indicated that this language allows a very pleasant implementation of algorithms which manipulate graphs, lists and arrays.

References

1. A. Rosenfeld, Picture Processing by Computer, Academic Press, 1969, (see also Computing Surveys, Sept. 1969).
2. C.T. Zahn, "A Formal Description for Two-dimensional Patterns," Proc. Intl. J. Conf. on A.I., Washington, D.C., May 1969, available as SLAC-PUB-538 from Stanford Linear Accelerator Center, Stanford, California 94305.
3. C.T. Zahn, "Region Boundaries on a Triangular Grid," Proc. 2nd Intl. J. Conf. on Pattern Recognition, Copenhagen, Denmark, August 1974, available as SLAC-PUB-1437.
4. H. Freeman, "Techniques for the digital computer analysis of chain-encoded arbitrary plane curves," Proc. Natl. Electronics Conf., 1961, pp 421-432.
5. C.T. Zahn and R.Z. Roskies, "Fourier Descriptors for Plane Closed Curves," IEEE Trans. Computers, Vol. C-21, No. 3, March 1972, pp 269-281.
6. C.T. Zahn, "Graph-theoretical methods for detecting and describing Gestalt clusters," IEEE Trans. Computers, Vol. C-20, No. 1, Jan. 1971, pp 68-86.
7. C.T. Zahn, "Using the Minimum Spanning Tree to Recognize Dotted and Dashed Curves," Proc. of an Intl.Computing Symp.(1973), A. Gunther et al (Editors), North-Holland Publ. Co., 1974.
8. D.E. Knuth, "The Art of Computer Programming: Sorting and Searching," Vol.3, Addison-Wesley, 1973.
9. R.O. Duda and P.E. Hart, Pattern Recognition and Scene Analysis, John Wiley, New York, 1973.
10. C.T. Zahn, "An Algorithm for Noisy Template Matching," Proc. IFIP Congress 1974, pp 698-701.
11. D.E. Knuth, "The Art of Computer Programming: Fundamental Algorithms," Vol.1, Addison Wesley, 1968.
12. C.T. Zahn, "Two-dimensional pattern description and recognition via curvaturepoints," SLAC Report No. 70, Dec. 1966.
13. R.L. Page, "A Minimal Spanning Tree Clustering Method[Z]," Algorithm 479, CACM, June 1974.
14. N.Wirth and C.A.R. Hoare, "A Contribution to the development of ALGOL," CACM, June 1966, pp 413-431.
15. N. Wirth, "The programming language PASCAL," Acta Informatica, Vol. 1, pp 35-63.
16. K. Jensen and N. Wirth, "PASCAL-User Manual and Report," Springer-Verlag Lecture Notes in Computer Science, Vol. 18, 1974.
17. C.A.R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," Acta Informatica, Vol. 3, pp 335-355.
18. C.A.R. Hoare, "Notes on Data Structuring" in Structured Programming by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, 1972.
19. D.E. Knuth, "Structured programming with go to statements," ACM Computing Surveys, Dec. 1974, pp 261-301.
20. A.J. Cook and L.J. Shustek, "MORTRAN2, a macro-based structured FORTRAN extension," presented at 10th IEEE Computer Society Intl. Conf. (COMPCON'75), available as SLAC-PUB-1527, Jan. 1975.
21. L.J. Shustek and C.T. Zahn, "Records and references in MORTRAN2," presented at ACM SIGNUM Workshop on FORTRAN preprocessors for Numeric Software, Jet Propulsion Lab, Pasadena, Calif., Nov. 1974.