A non-recursive technique for recreating a digraph from its K-formula representation

C. Bays

University of Oklahoma, Norman, Oklahoma

The K-formula notation is a compact representation introduced by Berziss (1971) and is a convenient method by which we may represent a digraph as a linear string while retaining all pertinent information associated with the digraph. Moreover the K-formula representation may be stored in a computer very compactly, requiring about half as much space as a full-blown list representation.

We may define the K-formula notation as follows. Given a digraph with node symbols a and b, then the following are true:

1. A single node symbol (e.g. 'x') is a K-formula.
2. We may write the K-formula *xy if and only if there exists a path from x to y.
3. The leftmost node symbol in a K-formula is called the "leading node" of the K-formula.
4. If a and b are K-formulas, then *ab is a K-formula if and only if there exists a path from the leading node of a to the leading node of b.

When a digraph contains a node, x, from which a path exists to all other nodes (a 'rooted' digraph), then it can be represented by a single K-formula with x as the leading node. A K-formula with n asterisks contains n + 1 node names and represents a digraph with n arcs. Furthermore, if a node has p arcs emanating from it, the K-formula will contain no more than p consecutive asterisks immediately preceding an occurrence of the node name in the K-formula.

This paper describes a process which will recreate a directed graph from any of its K-formula representations. The process involves a one-pass scan, from right to left, over the K-formula. At the termination of the scan, the graph will have been created and, depending upon the particular data structure used, may be either written out (or stored) as a set of ordered pairs, which have been created in a tree form, or manipulated internally as a list structure.

1.1. Terminology

Let i be a pointer to the K-formula entry being processed, where Ki is the ith K-formula entry (either a node name or an asterisk) in a sequential string of length n. A stack, W, is utilised during the process. W will contain nodes from the K-formula; these nodes as they are placed in stack W will be referred to by the letter Q. Associated with each node Q is a pointer, s, that points to the next lower Q entry in stack W. (The s-pointer of the first entry in W is set to λ, the null pointer). Also associated with each Q entry is a series of zero or more r-pointers emanating from it to other nodes. (The r-pointers actually form the arcs of the digraph). Let p point to the top node in stack W. Then, for example, node Qp may contain r-pointers r1p, r2p, .., rmip which point to other nodes in the digraph. (Note that each Q-entry will contain a pointer to its set of r-pointers, but this fact is not really germane to the presentation of the algorithms). Let x be a pointer that will always point to an available space for storing a new Q-entry when it is created. The basic method is given below.

Algorithm A

Build the graph as a tree

Step 1. set i ← n, p ← λ

Step 2. if Ki is a node,

   value of Qs ← value of Ki

   set s ← p, p ← x, go to step 4

Step 3. (Ki must be an *)

   add an r-pointer to Qp that points to location s

   set z ← sp, set sp ← sz proceed to step 4

Step 4. i ← i − 1; if i = 0 exit the algorithm

   otherwise, go to step 2.

3. An example

An example of the application of algorithm A is given below. Consider the digraph shown in Fig. 1. This digraph may be represented by several K-formulas, one of which is given as:

**ab****ccab

When we apply algorithm A to the above K-formula, the state of the structure at the conclusion of step 4 for i = 1 and i = 0 is given in Fig. 2.

*Present Address: Department of Mathematics and Computer Science, University of South Carolina, Columbia, S.C. 29208, USA.
Blocks represent nodes; r-pointers are represented by arrows emanating from the right side of blocks, and s-pointers point downward from the bottom of the blocks that are currently in stack $W$.

After the exit from algorithm A, we may traverse the resulting tree and write out the graph as the set

\[ \{ \langle a, c \rangle, \langle c, b \rangle, \langle c, a \rangle, \langle c, c \rangle, \langle a, b \rangle \} . \]

4. Representation of a digraph as a list

To scan the $K$-formula and at the same time give the true internal list representation of the digraph, we must eliminate the duplication of nodes that is caused by algorithm A. This can be accomplished by replacing stack $W$ with a list structure which, however, will retain the stack-like feature of having a ‘top’ pointer, and will operate on a first in, last out basis.

4.1. More terminology

Let $L$ represent the list. (As before, $p$ will point to the ‘top’ of the list). Associated with each node $Q_j$, in addition to its r-pointers, will be a series of zero or more s-pointers. These s-pointers will be arranged in a stack. Thus, node $Q_j$ will have associated with it pointers $r_1, r_2, \ldots, r_m$ and pointers $s_{1,j}, s_{2,j}, \ldots, s_{t,j}$ where $t_j$ points to the current top of the s-pointer stack for node $Q_j$. Hence, node $Q_j$ will have stored with it a pointer $t_j$ that points to the top of its individual S-stack. (Of course, its S-stack may be empty, in which case $t_j$ will point to location $j$).

If a doubly linked memory pool is used for available storage, and a hashing method is used to store node names and check for duplicates, the time spent searching for duplicate node names may be kept to a minimum (Bays, 1973). In fact, seldom will more than one or two accesses be required.

Note that the s-pointers and r-pointers for node $Q_j$ each contain two fields; one field will point to an appropriate Q-node and the other will point to the next s or r pointer in the s or r pointer list for $Q_j$. Let sv represent the field of an s-pointer that points to the next Q-node down the list. Let su represent the field that points to the next s-pointer in the stack of s-pointers for node $Q$. (A similar notation may be developed for the r-pointer list, but is not really pertinent to the explanation of the algorithm).

A diagram of a typical node, $Q_j$, should aid in understanding what the structure might really look like during an intermediate phase of processing a $K$-formula (Fig. 3). With the preceding explanation in mind, let us proceed with the algorithm.

Again, assume the $K$-formula is a sequential string of length $n$.

Algorithm B

Create the graph directly

Step 1. set $i = n$, $p = \lambda$

Step 2. (a) if $K_i$ is a node, search through the $Q$ nodes using the above mentioned hashing method, if desired) to find out if the value for node $K_i$ is already present. If it is, go to step 5, otherwise proceed to step 3.

(b) if $K_i$ is not a node (i.e. it is an *) go to step 6.

Step 3. (Assume $x$ is the location in which we shall create node $Q_x$.)

(a) value of $Q_x$ ← value of $K_i$
(b) retrieve an s-pointer from an available list.

Let $y$ point to this s-pointer
(c) $sv_y ← x$, $sv_y ← p$
(d) $t_x ← y$, $p ← x$, go to step 4

Step 4. set $i ← i - 1$; if $i = 0$, exit the algorithm, else go to step 2

Step 5. (Assume $x$ is the location we found that contained the value of $K_i$)

(a) retrieve an s-pointer from an available list.

Let $y$ point to this s-pointer
(b) $sv_y ← t_x$, $sv_y ← p$
(c) $t_x ← y$, $p ← x$; go to step 4

(Note that if we initialise all $t_j$ to $j$, step 3 may be omitted, and step 2 can go to step 5 unconditionally. This assumes that the hashing method we used will return either a pointer to a duplicate node value, or an empty space. In this case, step 3(a) will be performed redundantly with no harm done except for the time wasted).

Step 6. (a) set $z ← sv_p$

(b) create an r-pointer for $Q_p$ that points to location $z$
algorithm B can get quite complex, no unnecessary space is ever used up, with the exception of the \( t \)-pointers and the single remaining \( s \)-pointer.

Note that if the graph given in Fig. 1 had been written out as any set of \( K \)-formulas, for example
*\( ab, *ac, *ca, *cc, *cb \) (the arcs written as \( K \)-formulas) or
*\( ab, *a**ccab, *cb \)
e tc.

then if we treat a comma as a delimiter, which is ignored during the scan, Algorithm B will still produce the graph. Upon termination of the scan, there will remain several \( s \)-pointers, which may be returned to free storage, if desired. However, if it is known for a given graph that its \( K \)-formula is minimal, and the \( K \)-formula is given as \( K_A, K_B, K_C, \ldots \) etc. then the \( s \)-pointers that remain after the scan will link together the leading nodes of each subformula, with pointer \( p \) pointing to the leading node of subformula \( K_A \).

5.1. Storage requirements of a \( K \)-formula
A useful aspect of the \( K \)-formula representation is the compactness with which one may store a digraph. This is best explained by an example. Consider again the \( K \)-formula:
*\( *ab****ccab \)

which represents the digraph of Fig. 1.

If we define a ‘\( K \)-node’ as a node name together with the number of asterisks immediately preceding it, then we can rewrite the above \( K \)-formula as:

\[
2 \quad 0 \quad 3 \quad 0 \quad 0 \quad 0 \\
a \quad b \quad c \quad c \quad a \quad b
\]

where each number, together with the letter below it, constitutes a \( K \)-node.

Now, suppose we wish to represent a digraph which contains, say, 30 different nodes, with an average of two arcs and no more than seven arcs emanating from each node. A node will have information associated with it, but stored separately in a linear table somewhere.

Hence, since the maximum number of asterisks preceding each node in the \( K \)-formula is seven, we can represent each \( K \)-node with eight bits—five for the table address of the node information and three for the number of asterisks preceding the node name. Since we have 30 nodes and an average of two arcs emanating from each node, we have a total of 60 arcs, which give us 61 \( K \)-nodes, each one eight bits long, for a total of 488 bits.

If we were to represent this digraph as a list, each node would again require five bits for its table address and six bits for the address of its pointer list. (There are 60 pointers). Each pointer would be composed of a minimum of 11 bits, five for the node pointer and six for a pointer to the next pointer in the given node’s pointer list. The total number of bits required for this structure is 990.

With regard to information density, the above example should illustrate the efficiency with which we can represent a digraph using the \( K \)-formula notation.

References