Short Note

A Note on Computing Precedence Functions

An algorithm for computing the precedence functions of a given precedence matrix is presented. This algorithm is based on the parallel traversal of digraph representing the precedence matrix. It applies equally well to the computation of both simple precedence and operator precedence functions. Furthermore, it has pedagogical value in explaining the concept of longest path in the digraph context.

Introduction

Given a precedence matrix of relations between symbols of a programming language, there are several ways documented in the literature for computing its precedence functions. These are found to be unsatisfactory for various reasons. Floyd's original algorithm is based on the incremental adjustments of precedence functions with respect to the precedence matrix.[1] The incremental adjustments are enclosed in a loop, making the time efficiency of his algorithm an order of the matrix entries. Aho and Ullman document the other algorithm based on the digraph approach.[2] The precedence value of a symbol is simply stated to be the length of the longest path beginning from the node concerned, without further elaboration. Presumably one has to trace out every path from the node concerned in order to figure out which one is the longest path, and repeat this process for all the nodes in the digraph. This process is clearly very inefficient. The most efficient algorithm, out of all, certainly is Bell's solution[3] (see also Ref. 4). His method encodes the digraph of relations into a Boolean matrix. After computing the transitive closures of relations, the sums of row entries yield the precedence functions. Unfortunately, this algorithm still produces the precedence functions when there are none, as the following example shows. Suppose the following precedence matrix is given.

	a	*b*	*c*
a		>	<
b		<	>
c			

Applying Bell's algorithm, we have the following precedence functions:

	a	*b*	*c*
f	3	1	0
g	1	2	0

Clearly, these precedence functions are misleading as the precedence relations are cyclic. One therefore has to carry out the consistency test between the precedence functions so produced and the original precedence matrix.

We describe in what follows an alternative for computing the precedence functions based on the parallel computation approach. The non-existence of the precedence functions is automatically detected in the process of computing them.

Algorithm

The algorithm for computing the precedence functions has two phases. During the first phase, a digraph is constructed from the given precedence matrix. In the second phase, a parallel traversal of digraph is carried out for assigning a precedence value to each node. The resulting precedence values form the precedence functions. The following is a detailed sketch of the algorithm:

(1) Create two nodes, f_a and g_a, for each symbol including \$. (Following Ref. 2, \$ is used as the begin and end of the input symbol.)
(2) Form a digraph as follows: If $a > b$, place a directed link from f_a to g_b; If $a < b$, place a directed link from g_b to f_a; If $a = b$, place an undirected link between f_a and g_b. (Strictly speaking, the last precedence relation turns the digraph into a weak digraph.)
(3) Assume all nodes have zero precedence values to begin with. Starting from $f_\$$ and $g_\$$, move backwards along the directed links in parallel. When a node f_j is reached, its precedence value is compared with that of the node f_i just moved from. If the precedence value of f_j is greater than that of f_i, then do nothing. Otherwise, the precedence value of f_j is updated such that it is one greater than that of f_i. The new precedence value is then propagated to its companions connected by undirected links, if any. This parallel process is repeated until the computation converges or a node is assigned a precedence value equal to the number of nodes in the digraph. The former indicates the existence of precedence functions, and their values are the precedence values in the digraph. The latter signifies the contrary.

We can prove the correctness of this algorithm informally in the following. Since the bottom-up parsing traces out the canonical derivations in reverse, the longest path from any node will not exceed the distance between the node $f_\$$ or $g_\$$ and the node concerned. By starting the parallel traversal of digraph from $f_\$$ and $g_\$$, we are guaranteed to find the longest paths of all nodes, if any. When a node is reached via a directed or undirected link, if the new precedence value is greater than the old one, it means that the new path is longer than the old path being replaced, and vice versa. By this way, the minimum biggest precedence value, and hence the longest path, is always assigned to a node, as the digraph is traversed in parallel. If the graph is acyclic, the traversal will terminate when all sources are reached. As the precedence value assigned to each node equals to the longest path beginning from the node concerned, the precedence functions so computed are therefore consistent with the precedence matrix. If, on the other hand, the graph is cyclic, no precedence functions will exist, and the algorithm will fail to find ones when a precedence value equalling to the number of nodes in the digraph is assigned to a node.

Example

We now give an example to demonstrate how the algorithm works. The following precedence matrix of operator-precedence relations is adapted from:[2]

	+	*	(	)	id	\$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
\$	<	<	<		<	

The digraph constructed according to the algorithm is shown in Fig. 1.

The numbers written beside the nodes are the precedence values. By tabulating them in the usual way, we have the precedence functions.

	+	*	(	)	id	\$
f	2	4	0	4	4	0
g	1	3	5	0	5	0

Concluding remarks

This note arose from the pedagogical need in explaining the construction of precedence functions to students. The algorithm described in this note is based on the parallel traversal of digraph representing the precedence matrix. One could imagine that the way the parallel traversal works is similar to sending cars down the directed links. If all directed and undirected links are one and zero units long respectively, and all cars travel at the same speed from $f_\$$ and $g_\$$, the latest car arriving at a node must have travelled the longest distance. This analogy offers an intuitive explanation on why the algorithm works.

The parallel algorithm described in this note is applicable to the computation of both simple precedence and operator precedence functions.

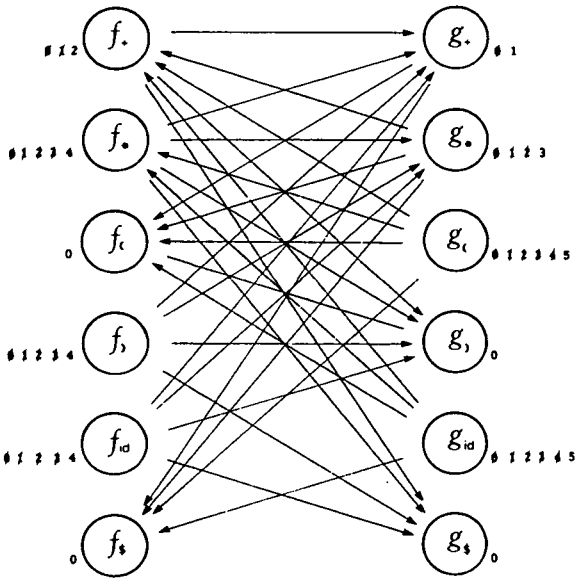

Figure 1. Digraph constructed according to the algorithm.

Acknowledgement

This work was supported under RGC Grant 05-143-105.

M. C. ER
Department of Computing Science,
The University of Wollongong, P.O. Box 1144,
Wollongong,
New South Wales 2500,
Australia

References

1. R. W. Floyd, Syntactic analysis and operator precedence, *Journal of ACM* **10**, 316–333 (1963).
2. A. V. Aho and J. D. Ullman, *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts (1977).
3. J. R. Bell, A new method for determining linear precedence functions for precedence grammars, *Communications of the ACM* **12**, 567–569 (1969).
4. D. Gries, *Compiler Construction for Digital Computers.* Wiley, New York (1971).

Received January 1982