4C-6

Detection of Cyclic Tree Structures

Martin Nilsson, Hidehiko Tanaka, Tohru Moto-oka
Information Engineering Course, The University of Tokyo

Abstract: Programs which process tree structures usually cannot handle cyclic trees. This paper describes some new, very simple, and efficient algorithms for detecting cyclic trees. Traversed structures do not have to be modified. Tail recursion optimisation can be used. The overhead for non-cyclic structures is very small. Unification is discussed as an application.

The Basic Method

Safe algorithms for cyclic trees are important and have received much attention, especially relating to Prolog unification. These algorithms are often complicated and specialised for particular applications.

Our methods for cycle detection are based on cycle detecting algorithms for lists (Knu 81), which we generalise to handle trees. The main idea is the following:

> Suppose we traverse trees in left-to-right, depth-first order. If we are walking down a path from the root of the tree, and encounter a node *already seen before on this path*, we have found a cycle. Thus we can use a list detection algorithm on this path. Termination follows from the termination of the list algorithm.

Note that the if the traversal is implemented by a recursive procedure, such as the one below, the path is available in the procedure argument stack during execution. As an example, we will generalise *Floyd's cycle detection algorithm* for lists (Knu 81). Floyd's algorithm keeps two pointers into the list. They are initially the same, but on every iteration, one of them moves one step forward, while the other one moves two steps forward. If the pointers become equal, a cycle has been found. A cycle will be found on the first repetition of the cycle.

Let us assume that the procedure argument stack can be referred to as an array, *stack*, with the stack pointer represented by a variable, *top*. This stack starts from position one, when *traverse* is first called. We also assume that data other than arguments (return addresses, frame pointers, etc) are made "invisible" by some method, e.g. address calculation.

Instead of letting Floyd's slow pointer step one step, and the fast pointer two steps on every iteration, we let them step a half, and one step, respectively. Then, the fast pointer will be the current argument to traverse. The slow pointer will be *just in the middle of the argument stack*. Floyd's algorithm for cyclic trees becomes

```
traverse(x)
{  if leaf(x) then process(x)
   else if x = stack[top/2]
     then cycle_detected;
   else {
      traverse(left(x));
      traverse(right(x));
   }
}
```

Fig. 1 Cycle detector

Variations

The other algorithms in (Knu 81) can also be implemented in a slightly more complicated way. *Brent's algorithm* is very similar to Floyd's algorithm, but allows easier tail recursion optimisation. In this way, very little stack space (logarithmic in the depth) will be consumed when we walk down a *right branch*. This is practical, since tree structures in many computer languages (e.g. Lisp and Prolog) branch more deeply to the right than to the left.

If most trees are shallow, one way to increase the efficiency for non-cyclic structures is to delay detection tests until a certain depth is reached. Another method is to use a fast procedure with a simplified heuristic detection algorithm. If it detects something which could be a cycle, a non-heuristic, slower procedure takes over.

The procedures in fig. 2 implement a generalised version of Brent's algorithm: The procedures optimise tail recursion, and use a fast heuristic test to find cycles. This algorithm does not use cyclic list detection on the *path*, but on the sequence of tree nodes in the order they are seen during traversal. In rare cases, some shared subtrees may falsely be "detected" as a cycle, so a second cycle check is necessary. The main idea behind Brent's algorithm is to number the nodes

```
traverse(x)
{   check := EMPTY;
    d := 1;
    traverse1(x);
}

traverse1(x)
{LOOP:
    if terminal(x) then process(x)
    else if x = check then
        slow_traverse_instead
    else {
        d := d + 1;
        if power_of_2(d) then
            check := x;
        traverse1(left(x));
        x := right(x);
        goto LOOP;
    }
}
```

Fig. 2 Heuristic cycle detector

in the order they are seen. Every new node of order d is compared with the last node of order L(d), where L(d) is the least power of two, less than or equal to d. It can be shown that the algorithm terminates within 3n steps, where n is the number of nodes in the tree. The test *power_of_2(d)* can be computed very easily: It is equivalent to *d & (d-1) = 0*, where *&* denotes bitwise AND.

Application: Unification

The unification procedure in fig 3 is typical for Prolog implementations. The two arguments to *unify* are the two structures to match. The procedures *dereference*, *bind*, *variable*, and *constant* are subroutines which: looks up a variable binding; binds a variable; tests if its argument is a pointer; and tests if its argument is a constant, respectively. (Here, it is assumed that the second argument is stored in the stack position above the first argument.)

```
unify(x,y)
{   x := dereference(x);
    y := dereference(y);
    if variable(x) then {
        bind(x,y); return(true);
    } else if variable(y) then {
        bind(y,x); return(true);
    } else if constant(x) or constant(y) then {
        return(x = y);
    } else if x = stack[top/2] and
            y = stack[top/2 + 1] then {
        cycle_detected;
    } else {
        return(unify(left(x),left(y)) and
            unify(right(x),right(y)));
    }
}
```

Fig. 3 Cycle detecting unification

Related work and Concluding discussion

Published algorithms for cycle detection in trees are usually specialised for unification. Their disadvantages are mainly that they require substantial memory or time overhead, even when cycles are rare. Also, they often temporarily change the traversed tree, which thus cannot be a read-only structure. It is usually hard to optimise tail recursion for these algorithms.

From the same points of view, the described algorithms are quite efficient. By tail recursion, the memory complexity can be made logarithmic in the length of left branches in a path, compared to usually linear for the other algorithms. A disadvantage with our algorithms is that they usually do not detect cycles at the earliest possible moment.

Our algorithms can be generalised to not only detect, but also traverse cyclic trees. Traversal of cyclic tree structures is described in a forthcoming paper.

Acknowledgments

We are grateful for comments by Keiji Hirata and Hanpei Koike. The ideas reported were partly studied in Sweden at UPMAIL, under sponsorship by the Swedish National Board for Technical Development. This research was possible thanks to a generous scholarship given by the Japanese Ministry of Education.

References

(Knu 81)
Knuth, D.E.: "The Art of Computer Programming," vol. 2, Seminumerical Algorithms, 2nd ed., problems 3.1.6-7, p. 7, 517-518. Addison-Wesley, 1981.