Cyclic Tree Traversal

Martin Nilsson and Hidehiko Tanaka
The Tanaka Laboratory, Graduate School of Information Engineering,
Department of Electrical Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo-ku, Tokyo 113

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

Unification is discussed as an application.

1. Introduction

The procedure in fig. 1-1 is a simple procedure for traversing binary tree structures.

```
traverse(x)
{  if leaf(x) then process(x)
   else {
      traverse(left(x));
      traverse(right(x));
   }
}
```

Fig. 1-1 Non-cyclic tree traversal

A serious problem with this procedure is that if it is applied to a "cyclic tree" such as the one in fig. 1-2, i.e. a tree where a node points back to one of its ancestors, it will never terminate. In a real implementation, the program will either loop for ever, or stop when the program stack or some other memory area overflows.

Safe algorithms for cyclic trees are important and have received much attention, especially concerning Prolog unification (Col 81), (Fag 83), (Fil 84), (HS 84), (Knu 81), (Muk 83), (YKTM 84). The problem of traversing trees with cycles is a more general problem than the problem of finding cycles in sequences studied in e.g. (Knu 81) and (SeSz 79). This paper presents methods for traversing cyclic trees. Our methods are

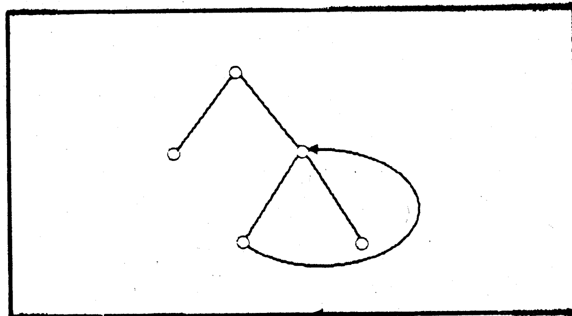

Fig. 1-2 Cyclic tree

based on cycle detecting algorithms for lists (Knu 81), which we generalise to detect and traverse cyclic 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.

We will introduce the general case of traversing cyclic trees by first studying detection of cyclic trees (NTM 86).

In our examples, we only use binary trees. Generalisation from binary trees to general trees is straightforward. We will use three primitive functions operating on a tree x: the Boolean *leaf(x)*, which says if x is a leaf, and *right(x)*, and *left(x)*, which extract the left and right subtrees.

For lists, cycles can be detected by saving all past nodes in a table, and compare every new node with the old ones. If equal, a cycle has been found.

For trees, we should only save all elements seen on the current path from the root. The recursive

procedure traverse in fig. 1-3 implements this algorithm for trees. Note that during the execution, all the past nodes on the current path will be available in the internal procedure argument stack during the execution.

```
traverse(x)
{ if leaf(x) then process(x)
  else if search_stack(x)
    then cycle_detected;
  else {
    traverse(left(x));
    traverse(right(x));
  }
}
```

Fig. 1-3 Inefficient cycle detector

The section in boldface is the addition for cycle detection. The procedure *search_stack(x)* compares *x* with previous arguments to *traverse* saved on the stack.

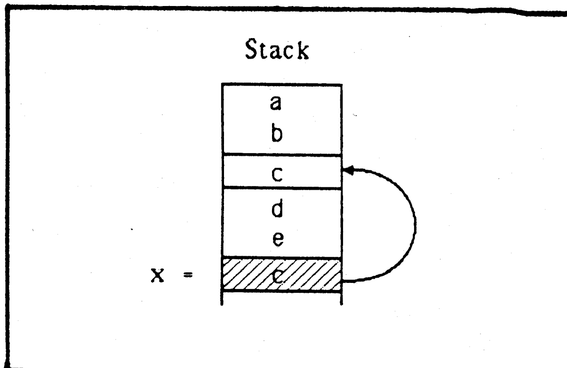

Fig. 1-4 *search_stack* finds past occurrences

As trees become deep, this algorithm becomes very inefficient. The search time increases in proportion to the depth, so the total time for a path of depth N will be $O(N^2)$. There are several more clever algorithms for finding cycles in lists (Knu 81), which can be generalised to handle trees: *Floyd's algorithm* keeps two pointers into the list. They are initially the same, but on every recursion, one of them moves one step forward, while the other pointer moves two steps forward. If the pointers become equal, a cycle has been found. A cycle will be found on the first repetition of an element.
Brent's algorithm remembers the latest 2^k:th element in the list. The subsequent elements are compared with this. Equality means that a cycle has been found. A disadvantage with this algorithm is that it takes longer before cycles are found. If the sequence repeats after N steps, Brent's algorithm stops within 3N steps.

Termination of Floyd's and Brent's algorithms is easy to show, and is given as an exercise in (Knu 81).

In section 2, we will generalise Floyd's and Brent's algorithms to detect and traverse cyclic trees. The idea is that trees can be traversed by trimming the stack when a cycle is detected, and continuing from there. Floyd's algorithm finds cycles after fewer steps, while Brent's algorithm is easier to use with tail recursion optimisation (TRO), shown in section 3. Some variants, including a heuristic detector, are described in section 4. Unification is discussed as an application in section 5. The relation to other work, discussion and conclusions are in sections 6 and 7.

2. Traversing cyclic trees

We will generalise Floyd's and Brent's algorithms to cyclic tree traversal, in depth-first, left-to-right order.

Floyd's and Brent's algorithms, as described in section 1, become directly applicable for detection of cyclic trees, if we replace the word "list" by "current path from the root of the tree."

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 is shown in fig. 2-1.

(When *top* is odd, the result of the division is not so important. When following Brent's algorithm strictly, the comparison is only performed for even values of *top*.)

```
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. 2-1 Floyd cycle detector

Let L(n) be the least power of two ≤ n. We get Brent's algorithm by replacing *stack[top/2]* in fig. 2-1 by *stack[L(top)]*. We will look up very few stack elements for comparison. If we put these stack elements in a separate small stack (of size at most $\log_2$ of the maximal depth), we can avoid looking into the internal stack. This makes it possible to optimise tail recursion for Brent's algorithm (see section 3).

Fig. 2-2 shows such a Brent algorithm for cyclic tree detection. The latest 2^k:th (i.e., the L(top):th) node in the current path is saved in a variable, *check*. The previous value of *check* is saved in the small stack by *pushsmall*. *check* should be initialised to a value which avoids accidental match with *x*.

```
traverse(x)
{  if leaf(x) then process(x)
   else if x = check
     then cycle_detected
   else if power_of_2(top) then {
     pushsmall(check);
     check := x;
     traverse(left(x));
     traverse(right(x));
     check := popsmall;
   } else {
     traverse(left(x));
     traverse(right(x));
   }
}
```

Fig. 2-2 Brent cycle detector

The test *power_of_2(top)* can be computed very easily: It is equivalent to *top & (top-1) = 0*, where *&* denotes bitwise AND.

The overhead in this program is almost only the comparison of *x* with *check*, and the computation and test of *top & (top - 1)*. This can be done in about four machine instructions per iteration. The very little data required for this overhead may be contained in fast memory, like registers or micro store.

We shall now see how we can extend the detection algorithms to traverse cyclic trees, i.e. continue after a cycle has been detected.

If we walk down a path from the root of a tree, and if we encounter a node already seen before on this path, we have found a cycle. Then, if we back up the path to just before the reoccurrence of any node, and go down the next branch in depth-first order, we will be able to continue traversing the rest of the tree without getting stuck in this cycle.

This is the central idea behind the traversal algorithms. The key point is the procedure *cycle_detected*. For mere detection, it is enough if this procedure simply calls an error handling routine, etc. For traversal, however, the procedure should trim the stack down to just before the first repetition of a node on the current path.

Such a procedure can in fact be implemented efficiently: Suppose a cycle is detected. That means that the stack top element *x* has already appeared in this path, and has been saved earlier in the stack. The distance between the first and the second occurences of x in the stack will be the cycle's period, and the first repetition of a

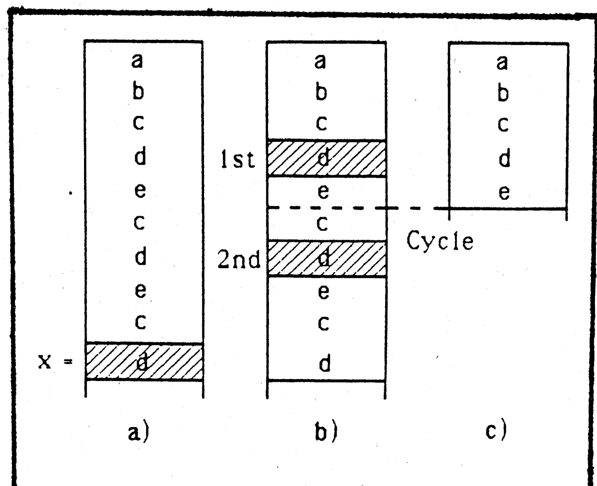

Fig. 2-3 The *cycle_detected* procedure

node in the path (i.e. the start of the cycle) must be between those occurences of *x*. In this way, we can easily find the beginning of the cycle, trim the stack back to that point, and continue with the next branch after the cycle. This method works identically for both Floyd and Brent type algoritms.

In fig. 2-3, a) shows the state of the stack when the cycle is first detected. In b), the first two occurrences of the stack top element are found, and in c) the stack has been trimmed to before the cycle.

3. Tail recursion optimisation

Since Brent's algorithm does not refer to the internal stack, we can easily implement TRO, as shown in this section.

Treé traversal programs are usually implemented in a more efficient way than *traverse* in fig. 1-1: The last call to *traverse* in the body can be replaced by a jump to the beginning, so we can write a non-cycle detecting version of *traverse* as in fig. 3-1.

```
traverse(x)
{LOOP:
   if leaf(x) then process(x)
   else {
      traverse(left(x));
      x := right(x);
      goto LOOP;
   }
}
```

Fig. 3-1 Tail recursive traversal

Stack space will only be consumed when we walk down a left branch, but not when we walk down a right branch. This is practical, since tree structures in many computer languages (e.g. Lisp and Prolog) usually branch to the right more often than to the left.

To adapt Brent's algorithm for TRO, we need to know the current depth in the tree. Before, the depth was given implicitly by the stack pointer, but now we need an explicit variable, *depth* for this purpose. The value of *depth* must be saved when we go down a left branch from a node, so the depth can be restored when it is time to go down the right branch.

When we return to a node after traversing its left subtree, the small stack must be trimmed back to the appropriate level. For this purpose, we keep the depth when the stack was last updated in a variable, *update_depth*.

From outside *traverse*, it is called with the depth argument = 1, and the variable *check* initially set to something which will not accidentally match *x*.

```
traverse(x, depth)
{LOOP:
   if leaf(x) then process(x)
   else if x = check then cycle_detected
   else {
      depth := depth + 1;
      if power_of_2(depth) then {
         update_depth := depth;
         pushsmall(check);
         check := x;
      }
      traverse(left(x), depth);
      while depth < update_depth do {
         check := popsmall;
         update_depth := update_depth/2;
      }
      x := right(x);
      goto LOOP;
   }
}
```

Fig. 3-2 TRO Cyclic tree traversal

Note that the procedure *cycle_detected* now also must remember to adjust the small stack to its appropriate size.

We will refer to this as our basic algorithm. The memory used will be linear in the maximum number of left branches in a path, and logarithmic in the maximum number of right branches.

4. Variations of the basic algorithm

We will describe some methods for increasing the efficiency, particularly for non-cyclic structures. Also, we describe a cycle detecting heuristic version of the basic algorithm.

The algorithm in fig. 3-1 can be reduced further: There is only one recursive call to *traverse* in

if the stack is not empty, execution must continue at the point just after this recursive call. If the stack becomes empty, execution is finished. The algorithm can be implemented directly as a loop, without any procedure calls, if we explicitly use primitives *push* and *pop* for saving and fetching past nodes on the stack:

```
LOOP:
    if leaf(x) then {
        process(x);
        x := pop;
        if not stack_empty then {
            x := right(x);
            goto LOOP;
        }
    } else {
        push(x);
        x := left(x);
        goto LOOP;
    }
```

Fig. 4-1 Open loop traversal

Since most trees will probably be rather shallow, one way to increase the efficiency for non-cyclic structures may be to delay detection tests until a certain depth is reached. In our basic algorithm this can be done by calling *traverse(x,depth)* with *depth* > 1.

Another method to lower the overhead is to use a fast procedure with a simplified heuristic detection algorithm. It cannot handle cycles, but when it detects something which could be a cycle, a cycle handling, slower procedure takes over. Our basic algorithm can be changed into such an algorithm by saving nodes, not according to depth in the tree, but according to the order in which they are seen during traversal. This means that we will not need so much stack handling in the program. We will surely find any cycle, although the algorithm sometimes "finds" cases which are not cycles, i.e. when a non-terminal subtree is shared by two subtrees.

A version of the basic algorithm which combines both these methods to detect cycles is shown in fig. 4-2.

If $N = 1$, it follows from Brent's algorithm that this algorithm stops within 3n iterations, where n is the number of nodes in the tree (shared

```
    d := N;
LOOP:
    if leaf(x) then {
        process(x);
        x := pop;
        if not stack_empty then {
            x := right(x);
            goto LOOP;
        }
    } else if x = check then
        slow_traverse_instead
    else {
        d := d + 1;
        if power_of_2(d) then check := x;
        push(x);
        x := left(x);
        goto LOOP;
    }
}
```

Fig. 4-2 Heuristic cycle detector

5. Application: Unification

Unification with occur check should fail as soon as a cycle is found in any argument. After successful unification, the result should be traversed to check that it doesn't contain any cycle. For unification without occur check, we need to find a repetition of a pair of arguments, *(x,y)*, to consider it a cycle.

The described methods for cycle traversal are easily adapted for both kinds of unification. Without cycle detection, a *unify* procedure consumes at least 2 units of stack memory for descending a left branch. With cycle traversal similar to that in fig. 3-2, *unify* will consume 3 units.

An attractive alternative may be to combine a fast heuristic detector with a slower, cycle-handling unifier.

The unification procedure in fig. 5-1 shows a Brent type unification procedure. 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

```
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. 5-1 Brent type unification

6. Related work and Discussion

Several algorithms for detection of cyclic structures have been published. They generally fall into three different categories:

- Pointers are (temporarily) replaced in the structures.
- Pointers are marked by tag bits.
- Some past nodes are saved and compared to new nodes.

Pointer replacements require undoing after execution. If restore information is saved on the stack, TRO becomes less practical. The memory complexity becomes O(L+R), where L and R are the maximum number of left and right branches in paths in the tree. Main memory references are needed for pointer replacements, which penalises non-cyclic structures. Since structures have to be changed, these methods are hard to use with read-only memory, or for shared structures in a parallel processing environment. An advantage is that these algorithms detect cycles after very few steps.

Tag marking also needs O(L+R) memory units. Unless the tags are in a separate table, the properties of these methods become similar to pointer replacement schemes. A separate tag bit table is uncomfortable because of its size and time for initialization.

The presented cyclic tree handling algorithms are the only ones we know of the third kind. The TRO cyclic tree traversal uses O(L + log(L+R)) memory units for non-cyclic structures. If there is a cycle on depth d, it will be discovered within depth 3d on the same path. The disadvantage is that these algorithms will be slow if there are many cycles in the structure.

For a real implementation, a good idea may be to combine fast heuristic detectors in hardware with slower, cycle handling algorithms in software.

7. Conclusions

The methods in this paper perform very well, regarding memory space and locality, low overhead for non-cyclic structures, and ability to handle read-only or shared structures. When cycles are very frequent, our methods could be combined with algorithms which have low overhead for cycles.

8. Acknowledgements

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.

In particular, we are deeply grateful to the late professor Tohru Moto-oka. By the outside world, he may perhaps be mostly remembered for his central role in the Fifth Generation Computer Project and countless other projects, but among his students and many other friends, he will always be remembered for his great generosity and kind heart.

9. References

(Col 81) Colmerauer, A.: "Prolog and Infinite Trees". In Clark, K. and Tärnlund, S.-Å.: (eds.): "Logic Programming." Academic Press, 1982.

(Fag 83) Fages, F.: "Note sur l'unification des termes de premier ordre finis et infinis." In conf. proc. Dincbas, M. (ed.): "Programmation en logique," Perros-Guirrec, France, March 22-23, 1983.

(Fil 84) Filgueiras, M.: "A Prolog interpreter

working with infinite terms". In Campbell, J.A. (ed.): "Implementations of Prolog." Ellis Horwood, Chichester, 1985.

(HS 84) Haridi, S., Sahlin, D.: "Efficient implementation of unification of cyclic structures." In Campbell, J.A. (ed.): "Implementations of Prolog." Ellis Horwood, Chichester, 1985.

(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.

(Muk 83) Mukai, K.: "A Unification algorithm for Infinite Trees." In Bundy, A. (ed.): Proc. of the Int. Joint Conf. on Artificial Intelligence, August 1983.

(NTM 83) Nilsson, M., Tanaka, H., Moto-oka, T.: "Detection of Cyclic Tree Structures:" In The Japanese Information Processing Society: Proc. 32nd Nat. Japanese Conf. Information Processing, 4C-6.1986.

(SeSz 79) Sedgewick, R. and Szymanski, S.G.: "The Complexity of Finding Periods." In Proc. ACM Symp. Th. Comp. 11, p. 74-80. 1979.

(YKTM 84) Yuhara, M., Koike, H., Tanaka, H., Moto-oka, T.: "A Unify Processor Pilot Machine for PIE." In Proc. of the Japanese Logic Programming Conference '84, Tokyo, March 1984.