AN EFFICIENT GARBAGE COLLECTOR FOR GRAPH MACHINES

Ashoke Deb

The Oregon Graduate Center
19600 NW Walker Road
Beaverton, Oregon 97006 U.S.

UUCP address: ogcvax!ashoke

CSE TR 84-003

AN EFFICIENT GARBAGE COLLECTOR FOR GRAPH MACHINES

Ashoke Deb
The Oregon Graduate Center

I. INTRODUCTION

Most of the garbage collection algorithms use two possible techniques : 1) Reference counting and 2) graph marking. Reference counting technique makes use of a field in each node, which contains a count of how many arrows point to this node. Such a count is rather easy to maintain as a program runs, and whenever it drops to zero, the node in question becomes available. But reference counting technique can not be used if the graph is recursive i.e. their are cycles in the graph. On the other hand, graph marking algorithms makes use of a special field , called mark field, which is used to mark all the reachable nodes from the root node of the graph. After the traversal of the entire tree , the nodes which are not marked (or 'specially' marked -- as in the case where marking algorithm uses more than one color to mark nodes) - become available to the free list. Such marking algorithms work for cyclic graphs. But in order to find all the reachable nodes, it has to traverse the entire graph -- which is very expensive.

In this paper we take a different route to the same problem. The strategy used by the traditional marking algorithms may be characterized as the one which , given a graph G such that there exists subgraphs G_1 and G_2 and $G_1 \cap G_2 = \varphi$ and $G = G_1 + G_2$ and G and G_1 has common root, will find G_1 by marking all the reachable nodes from R and then find G_2 as $G_2 = G - G_1$. Then G_2 is made available to the free list.

Our strategy is to find G_2 directly by traversing the smallest possible subgraph G_3 if G such that $G_2 \subseteq G_3 \subseteq G$. Given a node N whose successors are possible

candidates for the garbage collection - i.e the outgoing edges of N are to be deleted, we find the graph G_2 as the collection of exactly those nodes which are exclusively reachable from N, but not reachable from any node which is not reachable from N . Assuming that as the computation progresses, the subgraphs to be deleted are smaller than the subgraph to be retained, this algorithm is claimed to be highly efficient . In section II, we give a set of definitions, a computing environment consisting of a mutator process , a collector process and a structure, called the garbage can, and the algorithms. Section III contains the proofs of correctness of the algorithms. In section IV , we will discuss some of the merits of the algorithms presented, and then we present the results of the simulation . The simulation results confirm the efficiency of the algorithms ,in comparison to the traditional marking algorithms which mark the non-garbage nodes in the main graph.

II. DEFINITIONS AND ALGORITHMS

Definition 1: A **graph** G is defined as a pair (N,E), denoted as G = (N,E), where N is the set of **nodes** , which includes a special node labeled **nil** , and $E \subseteq N_1 \times N$, is the set of **edges** ,where N_1 = N - {nil}.

Definition 2: A graph is called a **binary** graph if for each node $A \in N$, there are exactly two edges e_1 and e_2 such that e_1 = (A,B), and e_2 = (A,C), where B, $C \in N$. These two edges are called **left edge** and **right edge** of A, respectively.

A left edge (A,B) [right edge (A,C)] will be represented as $A \xrightarrow{L} B$ [$A \xrightarrow{R} C$].

Without any loss of generality, we will consider only binary graphs. In such a graph, a non-nil node A is represented by a six-tuple described as :

$$A : (C_A, E_A, l_A, \text{leftofA}, \tau_A, \text{rightofA}),$$

where

$A \xrightarrow{L}$ leftofA,

$A \xrightarrow{R}$ rightofA,

$l_A(\tau_A)$: a binary digit, initially 1,

E_A : the in-degree of A,

C_A : initially 0.

C_A, l_A, τ_A are used and modified by the algorithms, **colorgreen** and **colorred** , to be described later.

The computing environment:

The computing environment consists of (i) a rooted, possibly cyclic, graph structure - in the case of a graph machine, the root represents the entire expression to be evaluated, cycles may represent recursive expressions, and a node may have multiple input edges in case of shared common sub-expression; (ii) a process which manipulates the graph structure by changing the existing links

between nodes- such a process is called the mutator; (iii) a process which will find the nodes which are no longer accessible from the root node of the graph - and hence referred to as garbage nodes; such a process is called the collector.

We will view the **the mutator** to be a process which deletes an input edge to a node N_i. One or many such deletions may disconnect a subgraph, from the main graph, thus, possibly creating a set of garbage nodes.

A data structure, called **the garbage-can**, possibly of fixed size **s** , is maintained, which will save temporarily the node name N_i one of whose input edges has been disconnected by the mutator.

When the garbage-can is full, **the collector** process uses those node names N_i's, to find all the garbage nodes.

The size s of the garbage-can may be used to dictate the frequency of calls to the collector; but there are other variations of this activity which are also possible.

Fig. 1 below shows an abstract view of the system. **Delete-an-edge-from(M)-to(N)** will (i) save N to the garbage-can (ii) modify the appropriate field of the node M. It will not decrement E_N, which will be done by the collector.

Detectgarbagefrom(garbage-can[1..s]) will (i) use the entries of the garbage-can to detect garbage nodes in the graph, (ii) decrement E_N for each entry N in the garbage-can.

In the following , we will describe, in detail, the algorithm for Detectgarbagefrom.

The Detectgarbagefrom algorithm makes use of two coloring algorithms , **colorgreen** and **colorred** .

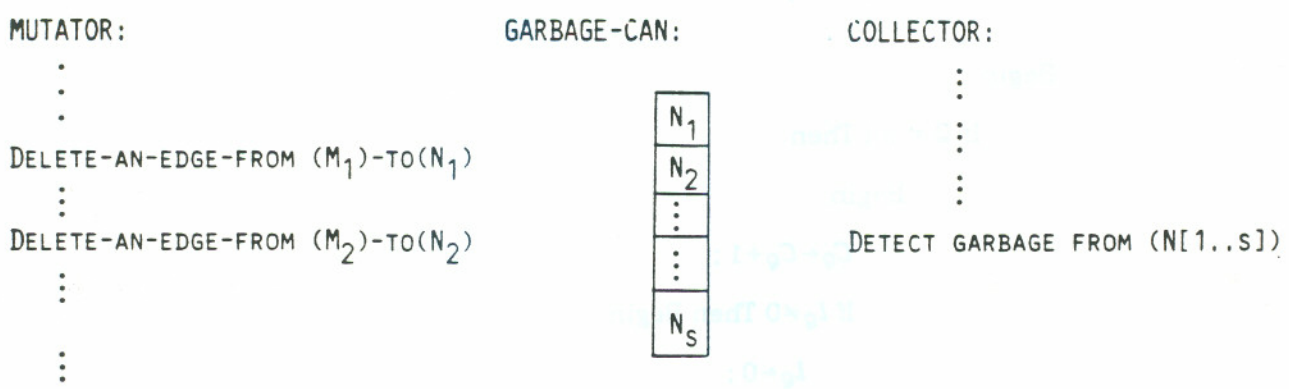


Fig. 1. An abstract view of the system.

The Detectgarbagefrom (N[1..s])

(* **comment**: Given a node N_i, this algorithm will color all nodes reachable from N_i such that all nodes exclusively reachable from N_i will have the color Green and the others which are not exclusively reachable from N_i, will have the color Red. Instead of using another field for color , the encoding $C_P = 0$ will represent the color Red and $C_P \geq 1$ will represent the Green color of a node P, respectively. Initial values of C_P is zero. *)

Begin

For i := 1 to s do colorgreen(N_i);

For i := 1 to s do colorred(green, N_i);

For i := 1 to s do $E_{N_i} \leftarrow E_{N_i} - 1$;

End.

The Green Algorithm : colorgreen (Q)

(* **comment**: Given a node Q, this algorithm colors all nodes P reachable from Q as Green, encoded as $C_P \geq 1$ *)

```
Begin
    If Q ≠ nil Then
        Begin
            C_Q ← C_Q + 1 ;
            If l_Q ≠ 0 Then Begin
                l_Q ← 0 ;
                colorgreen (leftofQ) ;
            End ;
            If r_Q ≠ 0 Then Begin
                r_Q ← 0 ;
                colorgreen (rightofQ)
            End
        End
End.
```

(* **comment**: The fields l_Q and r_Q are used here to ensure that each edge is traversed exactly once. Also note that after termination C_P value of each node P will be equal to the number of input edges to P reachable from the node Q *)

The Red Algorithm : colorred (Color, S)

(* **comment**: colorred will paint all nodes P red where P is not exclusively reachable from S. The encoding of the red color is $C_P=0$.*)

```
Begin
    If ( S≠nil and C_S≠0 ) Then
        Begin
            B <- (Color = Green and C_S=E_S) ;
If  B then
                Begin
                    If l_S≠1 Then Begin
                        l_S←1
                        colorred (Green, leftofS)
                    End;
                    If r_S≠1 Then Begin
                        r_S←1
                        colorred (Green, rightofS)
                    End;
                End;
            If  ¬B then
                Begin
                    C_S←0 ;
                    l_S←1 ;
                    colorred (Red,leftofS) ;
                    r_S←1 ;
                    colorred (Red,rightofS) ;
                End ;
        End
End.
```

Note : The purpose of $l_S \leftarrow 1$ and $r_S \leftarrow 1$ is to 'reset' back these bits in the node S if it gets colored Red so that we don't need to make another pass for resetting l_S, r_S. Similarly the encoding $C_S = 0$ meaning Red is chosen so that the nodes which will not be garbage collected will be reset back to their original condition .

Example 1

(see Fig.2)

In this diagram, terminating edges are assumed to be nil and 10 $\xrightarrow{R}$ 3, 13 $\xrightarrow{L}$ 6, 9 $\xrightarrow{L}$ 11. The edge 1 $\xrightarrow{R}$ 3 has been removed and now there is a virtual edge (shown as the dotted edge) pointing to 3.

Detectgarbagefrom will call colorgreen(3). The initial values of C and E of all relevant nodes are shown in Fig.3 . Fig.4 shows the essential trace of colorgreen(3) and the final values of C's.

Nodes	3	5	6	7	8	9	10	11	12	13
C	0	0	0	0	0	0	0	0	0	0
E	2	1	2	1	1	1	1	2	1	1

Fig.3

Node(P) visited	3	5	7	8	6	9	11	12	13	6	10	3
new C_P	1	1	1	1	1	1	1	1	2	1	2	

Fig.4

The colorred(Green, 3) is then called. Fig.5 shows the essential trace of colorred(Green, 3) and also shows their last Color values. If $C_P = E_P$ is maintained, then we depict that in the trace as G. If $C_P = 0$, then we depict that as R.

Node(P) visited	3	5	7	8	6	9	11	12	13	6	9	10	3	5	7	8
new C_P	G	G	G	G	G	G	R	R	R	R	R	R	R	R	R	R

Fig.5

For this example, one sees that there will not be any Green color node created, indicating that none of the nodes can be made available for the free list.

Example 2

(see Fig.6)

Example 2 is a modification of example 1 where $10 \overset{R}{\rightarrow} 6$ instead of $10 \overset{R}{\rightarrow} 3$.

Reader can convince himself that Detectgarbage in this example will leave the nodes 3,5,7 and 8 as Green colored ; and the nodes 6,9,10,11,12,13 as Red colored nodes.

III. DEFINITIONS AND PROOFS OF CORRECTNESS

Definition 3: An **1-chain (0-chain)** of length $n \geq 0$ from a node Q to a node A, A $\neq$ nil, denoted as $Q \overset{1^*}{\underset{n}{\rightarrow}} A$ ($Q \overset{0^*}{\underset{n}{\rightarrow}} A$), is defined to be an ordered sequence $(x_{A_0}, x_{A_1}, \ldots, x_{A_{n-1}})$, such that

$$x_{A_i} = L_i, \text{if } A_i \overset{L}{\rightarrow} A_{i+1} \text{ and } l_{L_i} = 1(0),$$

$$= R_i, \text{if } A_i \overset{R}{\rightarrow} A_{i+1} \text{ and } r_{L_i} = 1(0),$$

$$Q = A_0 \text{ and } A_n = A.$$

Definition 4: Two chains $P_1 = (x_{A_0}, x_{A_1}, \ldots, x_{A_{n-1}})$ and $P_2 = (y_{B_0}, y_{B_1}, \ldots, y_{B_{m-1}})$ are said to be **distinct** if $x_{A_i} \neq y_{B_j}$, for all i,j.

Definition 5: A chain $P = (x_{A_0}, x_{A_1}, \ldots, x_{A_{n-1}})$ is said to be **loop-free** if $x_{A_i} \neq x_{A_j}$, for all i,j.

The following simple lemmas follow directly from the Definition 3, which are presented here for the future reference.

Lemma 1: (i) $Q \overset{b^*}{\underset{0}{\rightarrow}} A \Rightarrow Q = A$;

(ii) $Q \neq A$ and $Q \overset{b^*}{\underset{n}{\rightarrow}} A \Rightarrow n > 0$; where b = 1 or 0.

Definition 6: Two chains P_1 and P_2, from a node Q to a node A, where $P_1 = (x_Q, x_{A_1}, \ldots, x_{A_{n-1}})$ and $P_2 = (y_Q, y_{B_1}, \ldots, y_{B_{m-1}})$, are said to be **end-distinct** if $x_{A_{n-1}} \neq y_{B_{m-1}}$.

Example : Consider a graph G represented by the following set of six-tuples :

$(C_A, 0, 0, B, 1, C)$

$(C_B, 2, 0, D, 1, E)$

$(C_C, 3, 1, E, 1, F)$

$(C_D, 1, 1, nil, 0, nil)$

$(C_E, 2, 1, nil, 1, C)$

$(C_F, 1, 0, B, 1, C)$

Some of the chains in the graph are

$$P_1 : \mathrm{A} \overset{0^*}{\underset{2}{\rightarrow}} \mathrm{D} = (L_A, L_B),$$

$$P_2 : \mathrm{A} \overset{1^*}{\underset{4}{\rightarrow}} \mathrm{F} = (R_A, L_C, R_E, R_C),$$

$$P_3 : \mathrm{A} \overset{1^*}{\underset{1}{\rightarrow}} \mathrm{C} = (R_A),$$

$$P_4 : \mathrm{A} \overset{1^*}{\underset{3}{\rightarrow}} \mathrm{C} = (R_A, L_C, R_E).$$

Also, P_3 and P_4 are end-distinct.

Definition 7: We define a set

$$\mathrm{CS[b,Q,A]} = \left\{ x_i \mid x_i : Q \overset{b^*}{\underset{n_i}{\rightarrow}} A, n_i \geq 0 \wedge x_i \, is \, loop-free \wedge end-distinct \right\}$$

Lemma 2 : Given any binary graph G, let

S_1 = CS[b,Q,A],

S_2 = CS[b,leftofQ,A],

S_3 =the largest subset of CS[b,rightofQ,A]

such that x $\in S_2$ and y $\in S_3$ implies that x and y are distinct.

Then $|S_1| = p_1 + p_2 |S_2| + p_3 |S_3|$, where p_2 (p_3) = 1, if l_Q (r_Q) = b; p_2 (p_3) = 0, otherwise; p_1 = 1, if Q = A; p_1 = 0, otherwise.

Proof : A b-chain x : $Q \overset{b^*}{\underset{n}{\rightarrow}} A \in S_1$ and n > 0

$\Leftrightarrow$ Either l_Q = b and $\exists$ a b-chain y $\in S_2$ such that x = (L_Q, y), Or, r_Q = b and $\exists$ a b-chain z $\in S_3$ such that x = (R_Q, z).

$\Leftrightarrow$ $|S| = p_2 |S_2| + p_3 |S_3|$, where p_2 (p_3) = 1, if l_Q (r_Q) = b.

If A = Q , then $Q \overset{b^*}{\underset{0}{\rightarrow}} Q \in S_1$ also. Hence, the theorem follows from the above two observations.

Q.E.D

In the following theorem, we show that, given a node S, after the execution of the algorithm **colorgreen(S)** , the field C_P of any node P is equal to the number of distinct input edges to P, each of which is in a chain from S to P.

Theorem 1: $\forall S \vdash \forall P \left\{ \left| CS[1,S,P] \right| = M[P,S] > 0 \wedge C_P = a_P \right\}$

colorgreen(S)

$\left\{ C_P = a_P + M[P,S] \wedge l_P = r_P = 0 \right\}$

Proof: For the sake of brevity, we will use the following notations:

I[P,S] $\equiv \left| CS[1,S,P] \right| = M[P,S] > 0 \wedge C_P = a[P,S]$

K[P,S] $\equiv C_P = a[P,S] + M[P,S] \wedge l_P = r_P = 0.$

Basis: [The last stage of recursion]

We need to consider two cases : (i) S=nil and (ii) $S \neq nil \wedge l_S = 0 \wedge r_S = 0$.

proof of (i):

We have to show that $\forall P(I[P,S] \wedge S = nil) \Rightarrow K[P,S]$.

Now, $S = nil$ => there does not exist any P such that $S \xrightarrow[n]{1^+} P$, n >= 0

=> I[P,S] is false.

Therefore, the theorem holds vacuously.

proof of (ii):

We have to show that

$\forall p \left\{ I[P,S] \wedge S \neq nil \wedge l_S = r_S = 0 \right\} C_S \leftarrow C_S + 1 \left\{ K[P,S] \right\}$.

Now, I[P,S] $\wedge S \neq nil \wedge l_S = r_S = 0$

$\Rightarrow \forall P, S \xrightarrow[n]{1^+} P,\ n = 0$

$\Rightarrow S = P \wedge M[P,S] = 1$ ---(from Lemma 1 and Lemma 2).

Clearly,

$$\left\{I[P,S] \wedge S \neq nil \wedge l_S = r_S = 0\right\}\ C_S \leftarrow C_S + 1\ \left\{S = P \wedge l_S = r_S = 0 \wedge C_S = a[P,S] + 1\right\}$$

holds.

Induction : Assume that the theorem holds for the n-th level of recursion. We have to show that it also holds for the (n+1)-th level of recursion.

It suffices to show that $\left\{I[P,S] \wedge S \neq nil \wedge (ls = 1 \vee r_S = 1)\right\}$ colorgreen(S) $\left\{K[P,S]\right\}$ holds.

From the assumption, we can very easily conclude that both of the following hold:

$$\left\{\left|CS[1,leftofS,P]\right| = M[P,leftofS] > 0 \wedge C_P = b_P \wedge l_S = x\right\}$$

first-if-then

$$\left\{\left\{x = 1 \Rightarrow (cp = b_P + M[P,leftofS] \wedge l_P = r_P = 0)\right\} \wedge (x = 0 \Rightarrow C_P = b_P) \wedge l_S = 0\right\},$$

and similarly,

$$\left\{\left|CS[1,rightofS,P]\right| = M[P,rightofS] > 0 \wedge C_P = d_P \wedge r_S = z\right\}$$

second-if-then

$$\left\{\left\{z = 1 \Rightarrow (cp = d_P + M[P,rightofS] \wedge l_P = r_P = 0)\right\} \wedge (z = 0 \Rightarrow C_P = d_P) \wedge r_S = 0\right\},$$

where

first-if-then ≡ If $l_S \neq 0$ then begin

$l_S \leftarrow 0$;

colorgreen(leftofS)

end.

and,

second-if-then ≡ If $r_S \neq 0$ then begin

$r_S \leftarrow 0$;

colorgreen(rightofS)

end.

Additionally, the following holds

$$\{I[P,S] \wedge S \neq nil \wedge l_S = x \wedge r_S = y\}$$

$C_S \leftarrow C_S + 1$

$$\{S = P \Rightarrow C_P = a_P + 1 \wedge S \neq P \Rightarrow C_P = a_P\}.$$

By using the above results and the fact that

$(M[P,S] > 0 \wedge M[P,leftofS] > 0) \Rightarrow (x = 1)$

and similarly,

$(M[P,S] > 0 \wedge M[P,rightofS] > 0) \Rightarrow (z = 1)$,

we conclude that

$$\{I[P,S] \wedge S \neq nil \wedge l_S = x \wedge r_S = y\}$$

$C_S \leftarrow C_S + 1$;

first-if-then;

second-if-then

$$\{C_P = a_P + p_1 + p_2 M[P,leftofS] + p_3 M[P,rightofS] \wedge (x = 1 \vee z = 1)$$

$$\Rightarrow (lp = 0 \wedge r_P = 0) \quad \wedge (ls = 0 \wedge r_S = 0) \wedge \quad (P = S \Rightarrow p_1 = 1) \wedge$$

$$(P \neq S \Rightarrow p_1 = 0) \wedge (x = 1 \Rightarrow p_2 = 1) \wedge (x = 0 \Rightarrow p_2 = 0) \wedge (z = 1 \Rightarrow p_3 = 1) \wedge$$

$$(z = 0 \Rightarrow p_3 = 0)\}$$

Finally, by using the fact that $P \neq S \Rightarrow (x = 1 \vee z = 1)$ and by applying Lemma

2, we get

$$\left\{I[P,S] \wedge S \neq nil \wedge l_S = x \wedge r_S = y\right\}$$

$C_S \leftarrow C_S + 1$;

first-if-then;

second-if-then

$$\left\{C_P = a_P + M[P,S] \wedge l_P = r_P = 0\right\}$$

Q.E.D

From the above theorem one observes that, given a graph where initial value of C_P, for any node P, is zero and $l_P = r_P = 1$, colorgreen(S) will (i) visit all the edges reachable from S exactly once, (ii) update C_P's to be equal to the number of input edges to P reachable from S and (iii) leave its mark on the edges travelled by setting l_P's and r_P's to zeros.

From the above, one can also observe that ,given a set of nodes S_1, S_2,...,S_k, the order in which colorgreen(S_1), colorgreen(S_2),..., colorgreen(S_k), are invoked is inconsequential to the total number of the edges travelled, the values of the C_P's, and the values of the l_P's and r_P's.

Next we will prove a property of the algorithm **colorred(color,S)** ,which shows that the algorithm colorred(color,S) resets the C_P values to 0 ,for all P's which are not **exclusively** reachable from S. Thus, nodes with C_P values equal to 0 are not garbage yet.

Definition 8 : We define a set

$$Z_S = \left\{P_i \mid S \xrightarrow[n_i]{0^*} P_i, n_i \geq 0\right\}$$

Theorem 2 : $\forall S \vdash \forall P \left\{ P \in Z_S \wedge color = c_S \wedge C_S = k_{1S} > 0 \wedge E_S = k_{2S} \right\}$

colorred(color,S)

$\left\{ ((c_S = red \vee (k_{1S} < k_{2S})) => C_P = 0) \wedge \neg(c_S = red \vee (k_{1S} < k_{2S}) => (\exists R, R \in Z_S \wedge P \in Z_R \wedge C_R = 0) => C_P = 0) \wedge l_P = 1 \wedge r_P = 1 \right\}$

Proof : For the sake of brevity, we will use the following notations:

A[P,S] $\equiv P \in Z_S \wedge color = c_S \wedge C_S = k_{1S} > 0 \wedge E_S = k_{2S}$

B[S] $\equiv c_S = red \vee (k_{1S} < k_{2S})$

D[P] $\equiv C_P = 0$

E[P,S] $\equiv \exists R, R \in Z_S \wedge P \in Z_R \wedge C_R = 0$

F[P,S] $\equiv (B[S] => D[P]) \wedge (\neg B[S] => (E[P,S] => D[P]))$

Therefore, we need to show that , for any node S and P,

{A[P,S]} colorred(color,S) {F[P,S]} holds true.

basis: [last stage of recursion]

There are three possible cases which have to be considered: (i) S = nil (ii) $C_S = 0$

(iii) $\neg B[S] \wedge l_S = 1 \wedge r_S = 1$.

proof of (i): S = nil => $\neg(\exists P, S \xrightarrow[n]{0^*} P, n \geq 0)$ => A[P,S] is false.

Therefore, the theorem holds vacuously.

proof of (ii): Also, $C_S = 0$ => A[P,S] is false.

proof of (iii): Let G[S] $\equiv \neg B[S] \wedge l_S = 1 \wedge r_S = 1$.

From the definition of Z_S, we have

$(A[P,S] \wedge l_S = 1 \wedge r_S = 1) => (P = S)$.

Also, since

E[S,S] => $C_S = 0$,

then

$(P = S \wedge G[S]) => (notB[S] => (E[P,S] => D[P])) \wedge l_P = 1 \wedge r_P = 1$.

Hence the theorem holds.

Induction: Assume that the theorem holds for the n-th level of recursion. We want to show that it also holds for the (n+1)-th level.

At the (n+1)-th level of recursion, since it is not the basis, we have the following condition:

$S \neq nil \wedge (C_S = k_{1S} > 0) \wedge \neg G[S]$

$\equiv S \neq nil \wedge (C_S = k_{1S} > 0) \wedge (B[S] \vee l_S = 0 \vee r_S = 0)$ is true.

Now it suffices to show that both (i) and (ii) below hold.

(i) $\{A[P,S] \wedge S \neq nil \wedge (C_S = k_{1S} > 0) \wedge B[S]\}\ colorred(color,S)\ \{F[P,S]\}$

(ii) $\{A[P,S] \wedge S \neq nil \wedge (C_S = k_{1S} > 0) \wedge \neg B[S] \wedge (ls = 0 \vee r_S = 0)\}$

$colorred(color,S)\ \{F[P,S]\}$

proof of (i): To prove (i), it suffices to show that the following holds:

$\{A[P,S] \wedge S \neq nil \wedge (C_S = k_{1S} > 0) \wedge B[S]\}$

$C_S \leftarrow 0;$

$l_S \leq 1;$

$colorred(red, leftof S);$

$r_S \leftarrow 1;$

$colorred(red, rightof S);$

$\{D[P] \wedge l_P = 1 \wedge r_P = 1\},$

since B[S] remains uneffected and since

$(B[S] \wedge D[P] \wedge l_P = 1 \wedge r_P = 1) \Rightarrow F[P,S].$

By assumption, we know that

$$\{A[P,leftofS]\}colorred(red,leftofS)\{c_{leftofS}=red \wedge F[P,leftofS]\},$$

and

$(c_{leftofS}=red \wedge F[P,leftofS]) => (D[P] \wedge l_P=1 \wedge r_P=1)$.

Similarly, we conclude that

$$\{A[P,rightofS]\}colorred(red,rightofS)\{c_{rightofS}=red \wedge F[P,rightofS]\},$$

and

$(c_{leftofS}=red \wedge F[P,leftofS]) => (D[P] \wedge l_P=1 \wedge r_P=1)$.

Finally,

$((C_S=0 \wedge l_S=1 \wedge r_S=1) \quad \wedge (P \in Z_{leftofS} => D[P]) \quad \wedge (P \in Z_{rightofS} => D[P])$

$\wedge l_P=1 \wedge r_P=1) \; => \; (D[P] \wedge l_P=1 \wedge r_P=1)$.

Therefore the theorem holds for (i).

proof of (ii): To prove (ii), it suffices to show that the following holds:

$$\{A[P,S] \wedge S \neq nil \wedge (C_S=k_{1S}>0) \wedge (\neg B[S] \wedge (l_S=0 \vee r_S=0))\}$$

first-if-then;

second-if-then;

$$\{(E[P,S] => D[P]) \wedge l_P=1 \wedge r_P=1\},$$

since, ¬ B[S] remains unaffected and since

$\neg B[S] \wedge (E[P,S] => D[P]) \wedge l_P=1 \wedge r_P=1 => F[P,S]$;

where

first-if-then ≡ If $l_S \neq 1$ then begin

$l_S \leftarrow 1$;

$colorred(green,leftofS)$; end;

second-if-then ≡ If $r_S \neq 1$ then begin

$r_S \leftarrow 1$;

$colorred(green, rightofS)$; end;

By assumption, we have that

$$\{A[P, leftofS] \wedge l_S = x\}$$

first-if-then

$$\{(x = 0 \Rightarrow F[P, leftofS]) \wedge l_S = 1\},$$

and similarly,

$$\{A[P, rightofS] \wedge r_S = x\}$$

second-if-then

$$\{(y = 0 \Rightarrow F[P, rightofS]) \wedge r_S = 1\}.$$

Since, by the definition of Z,

$P \in Z_S \Leftrightarrow (P=S \vee P \in Z_{leftofS} \vee P \in Z_{rightofS})$, and

$(P \in Z_S \wedge P \in Z_{leftofS}) \Rightarrow x = 0$, and $(P \in Z_S \wedge P \in Z_{rightofS}) \Rightarrow y = 0$, and by the initial condition,

$x \vee y = 0$,

we simply need to show that

(i) $(P=S \wedge l_S = 1 \wedge r_S = 1 \wedge E[P,S]) \Rightarrow (D[P] \wedge l_P = 1 \wedge r_P = 1)$,

(ii) $(P \in Z_S \wedge P \in Z_{leftofS} \wedge x = 0 \wedge F[P, leftofS] \wedge E[P,S])$

$\Rightarrow (D[P] \wedge l_P = 1 \wedge r_P = 1)$,

and similarly,

(iii) $(P \in Z_S \wedge P \in Z_{rightofS} \wedge y = 0 \wedge F[P, rightofS] \wedge E[P,S])$

$\Rightarrow (D[P] \wedge l_P = 1 \wedge r_P = 1)$.

proof of (i): Follows directly, since E[S,S] => D[S].

proof of (ii):

$(P \in Z_S \wedge P \in Z_{leftofS} \wedge x = 0 \wedge F[P,leftofS] \wedge E[P,S]) => E[P,leftofS]$.

Now,

$(F[P,leftofS] \wedge E[P,leftofS]) => (D[P] \wedge l_P = 1 \wedge r_P = 1)$.

Proof of (iii): is similar to the one above.

Q.E.D

IV. SOME EFFICIENCY MEASURES AND RESULTS OF SIMULATIONS

i) Cost Functions.

The best garbage collection routines known have an execution time essentially of the form $C_1N + C_2M$, where C_1 and C_2 are constants, N is the number of nodes marked, and M is the total number of nodes in the memory. Thus M - N is the number of free nodes found, and the amount of time required to return these nodes to free storage is $(C_1N + C_2M)/ (M - N)$ per node. Let $N = \rho M$; this figure then becomes $(C_1 \rho + C_2)/(1 - \rho)$. So if $\rho = 3/4$ i.e. if the memory is three -fourths full, it takes $3 C_1 + 4 C_2$ units of time per node returned to storage; when $\rho = 1/4$, the corresponding figure is only $1/3 C_1 + 4/3 C_2$. If one uses only the reference count technique, the amount of time per node returned is essentially a constant C_3, and it is doubtful that C_3 / C_1 will be very large. Hence we can see to what extent garbage collection is inefficient when the memory becomes full, and how it is correspondingly efficient when demand on memory is light.

Following the above analysis, as given in [1],let's see where the algorithm Detectgarbagefrom stands.

The algorithm presented here traverses the subset N_1 of nodes and marks (M - N) nodes as garbage. Hence the cost per node is $(C_1 N_1 + C_2 M)/(M-N)$. Let $N_1 = (M - N) + N_2$, where N_2 is the number of nodes traversed by the algorithm but are not garbage. If $N_2 << (M - N)$ then the cost per node is equal to $C_1 + C_2 /(1 - \rho)$, where $\rho = N/M$.

Also, there are many classes of computations where the rate of consumption of nodes is larger compared to the the deletions -- in which cases $M - N < N$, or $\rho > 1/2$.

Under this conditions, we demonstrate the following efficiency relation between Markgoodnodes and Detectgarbagefrom where

$$\text{Markgoodnodes} = (C_1 \rho + C_2)/(1 - \rho) \text{ and}$$

$$\text{Detectgarbagefrom} = C_1 + C_2 / (1 - \rho) .$$

From these equations, it follows that

$$\text{Markgoodnodes} - \text{Detectgarbagefrom} = C_1 (2\rho - 1)/(1 - \rho) .$$

Since $1 \geq \rho > 1/2$, Markgoodnodes - Detectgarbagefrom > 0 ;

and also as ρ tends towards 1, (Markgoodnodes - Detectgarbagefrom) tends towards a significantly large quantity.

ii) Stack or no stack .

The most interesting feature of garbage collection is the fact that while this algorithm is running, there is only a very limited amount of storage available which we can use to control our marking algorithm.

If one chooses to implement the Green algorithm and the Red algorithm using simple stacks, then obviously the simplicity of the implementation will make it run fast -- i.e. time efficient. Stack size is proportional to the length of the longest path in the tree -- and in the worst case, proportional to the size of the tree traversed. If it is the case that the tree to be deleted is much smaller than the tree to be retained -- or, the frequency of invocation of the Detectgarbagefrom algorithm is such that the above assumption is justifiable -- then a comparatively small stack area is only needed.

But the Green algorithm and the Red algorithm can also be implemented without using stacks. Such implementation would be similar to the techniques in

[2], where as the tree is traversed downward the structure of the tree is temporarily altered – by switching pointers; and as the tree is traversed upward the original tree is restored back.

iii) Simulation Results. Simulation was performed by randomly generating graphs of total nodes upto ten thousands, and then randomly modifying the links between nodes.

The size of the garbage-can was varied from 2 to 16. Each time the garbage-can was full, the garbage collector was invoked and also the total number of nodes visited, both during colorgreen and colorred phases, are counted. Then a conventional marking algorithm was used for garbage collection and the number of nodes visited was counted.

The relative performances of these two algorithms are shown in the accopanying plots. Continuous smooth lines were drawn through the actual points, to see the sensitivity pattern of the algorithms with respect to the amount of the garbage actually collected.

REFERENCES

1. Knuth,D.E., "The Art of Computer Programming",AW, 1973.

2. Schorr,H.,Waite,W.M.,"An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures", CACM,vol.10,No.8,1967 .

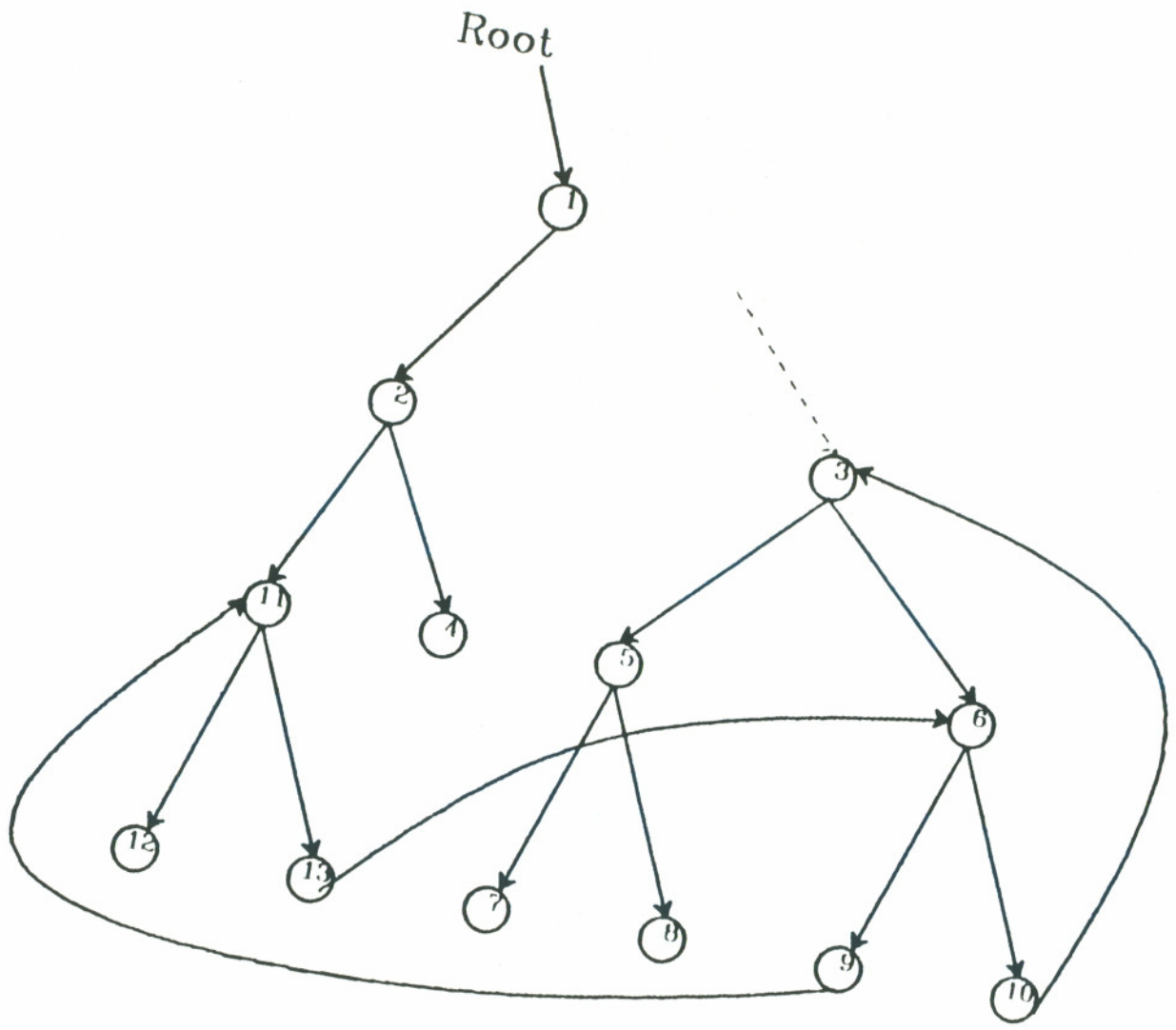


Fig. 2

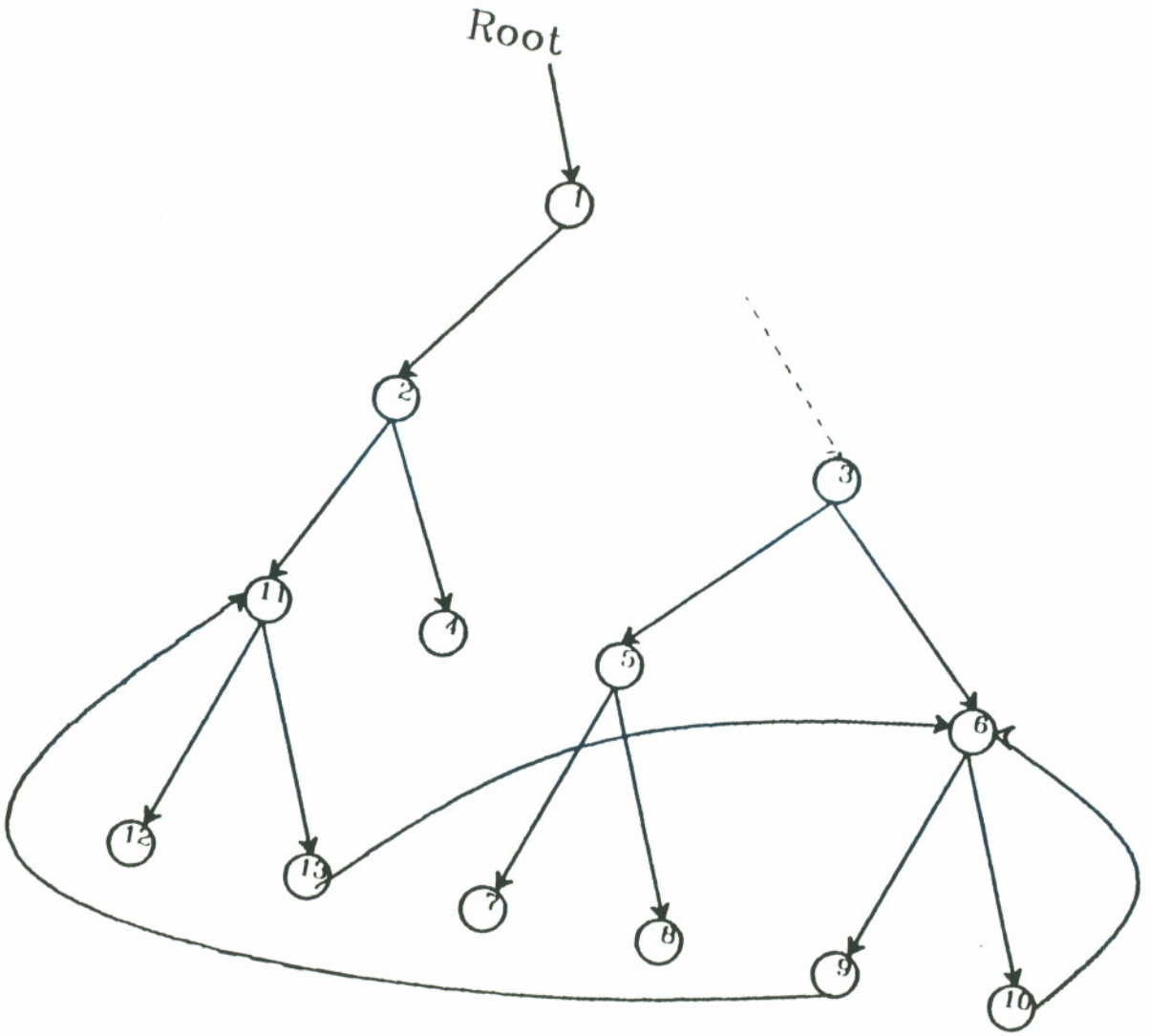


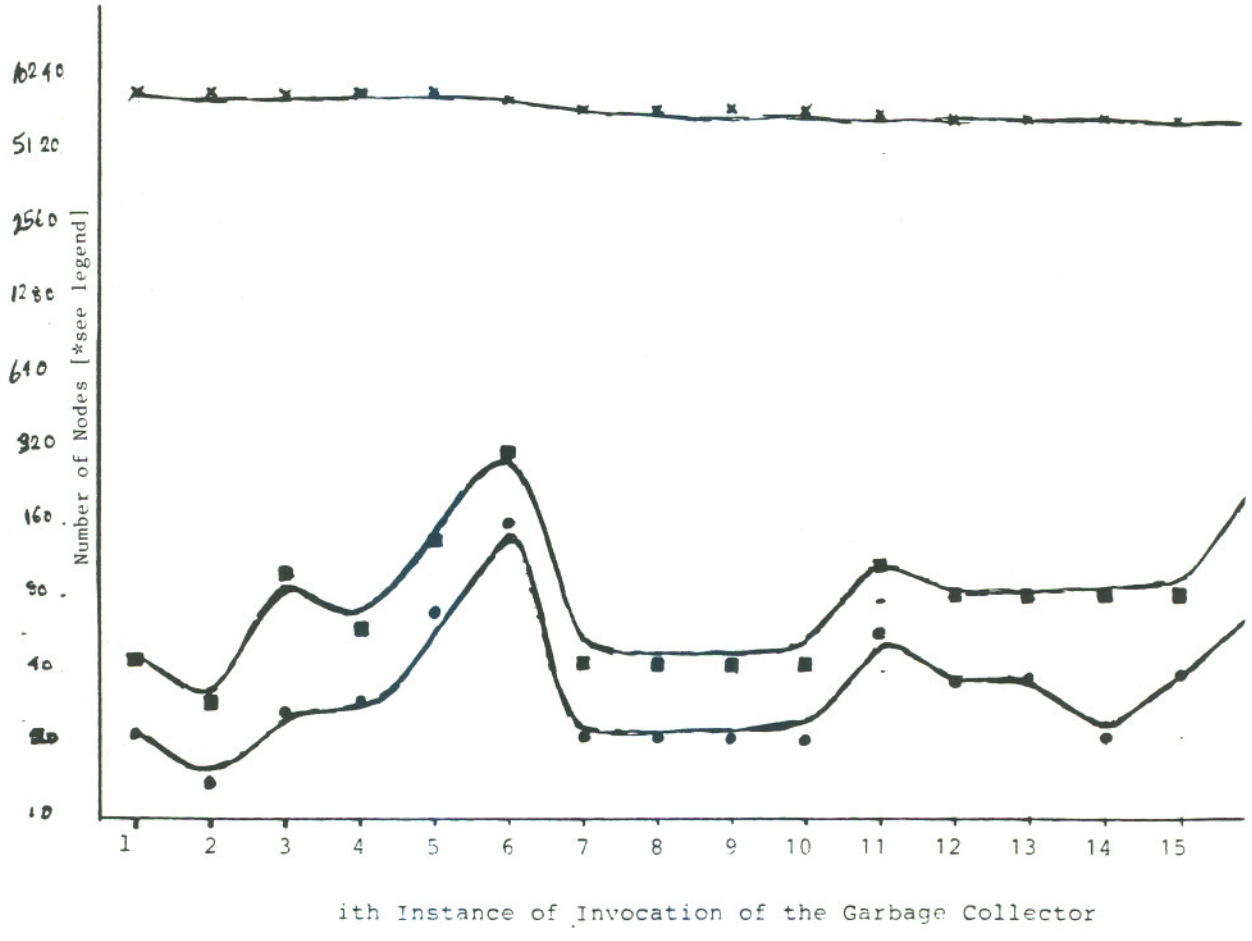
Fig. 6

LEGEND: Top Graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected.

Seed = 7774755
Size of Garbage-Can = 4

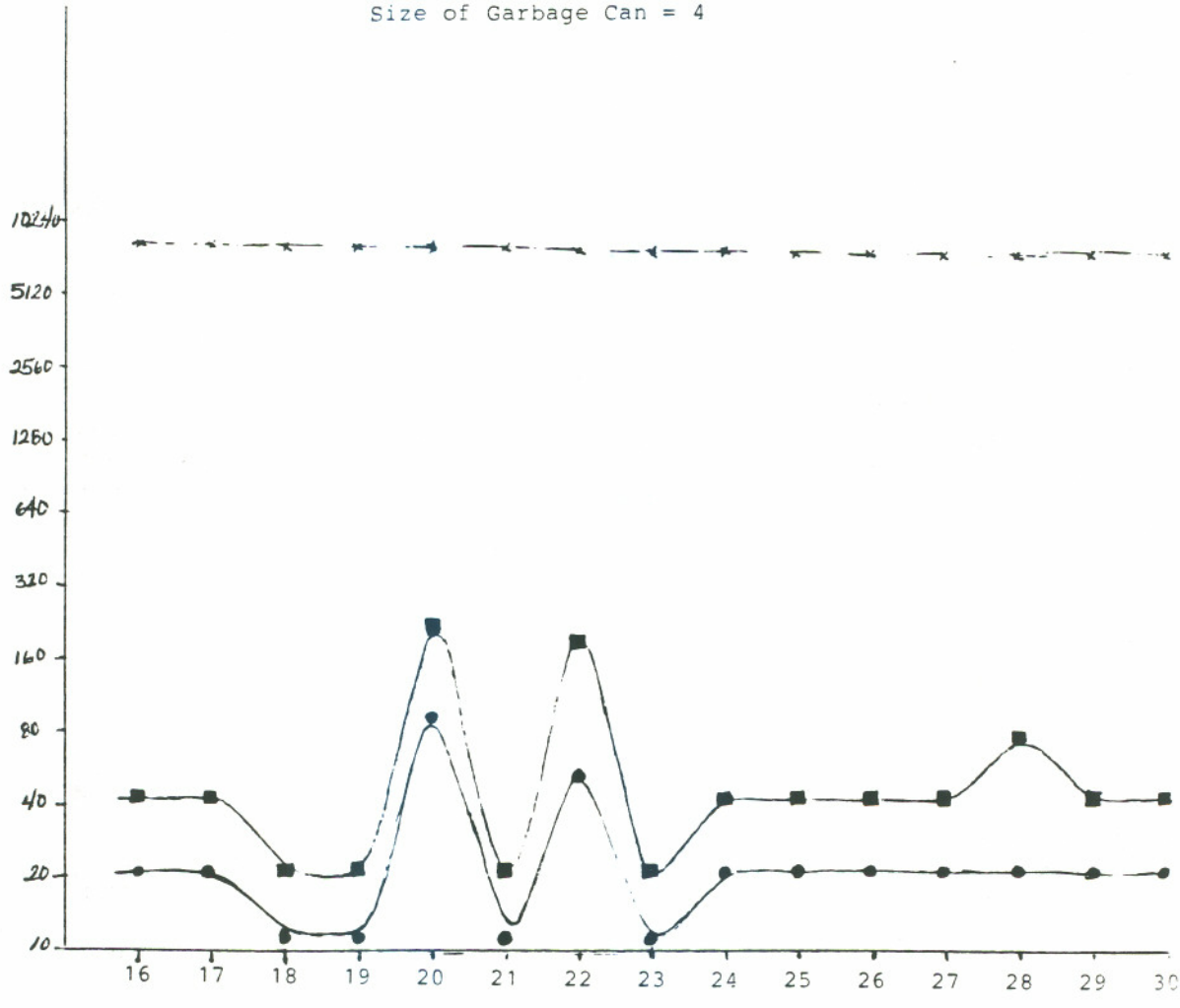


LEGEND: Top Graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected.

Seed = 7774755
Size of Garbage Can = 4



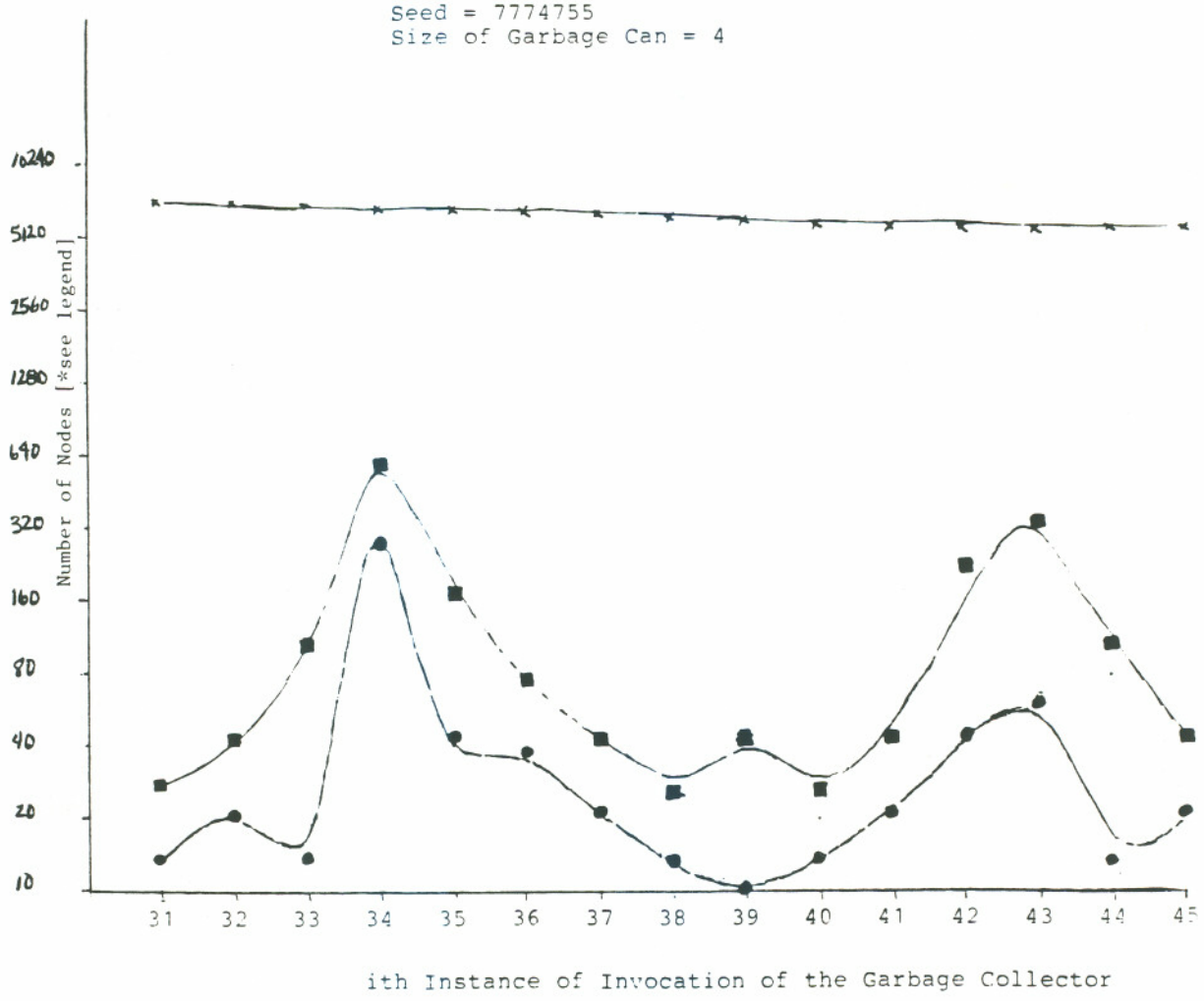
ith Instance of Invocation of the Garbage Collector

LEGEND: Top graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected

Seed = 7774755
Size of Garbage Can = 4

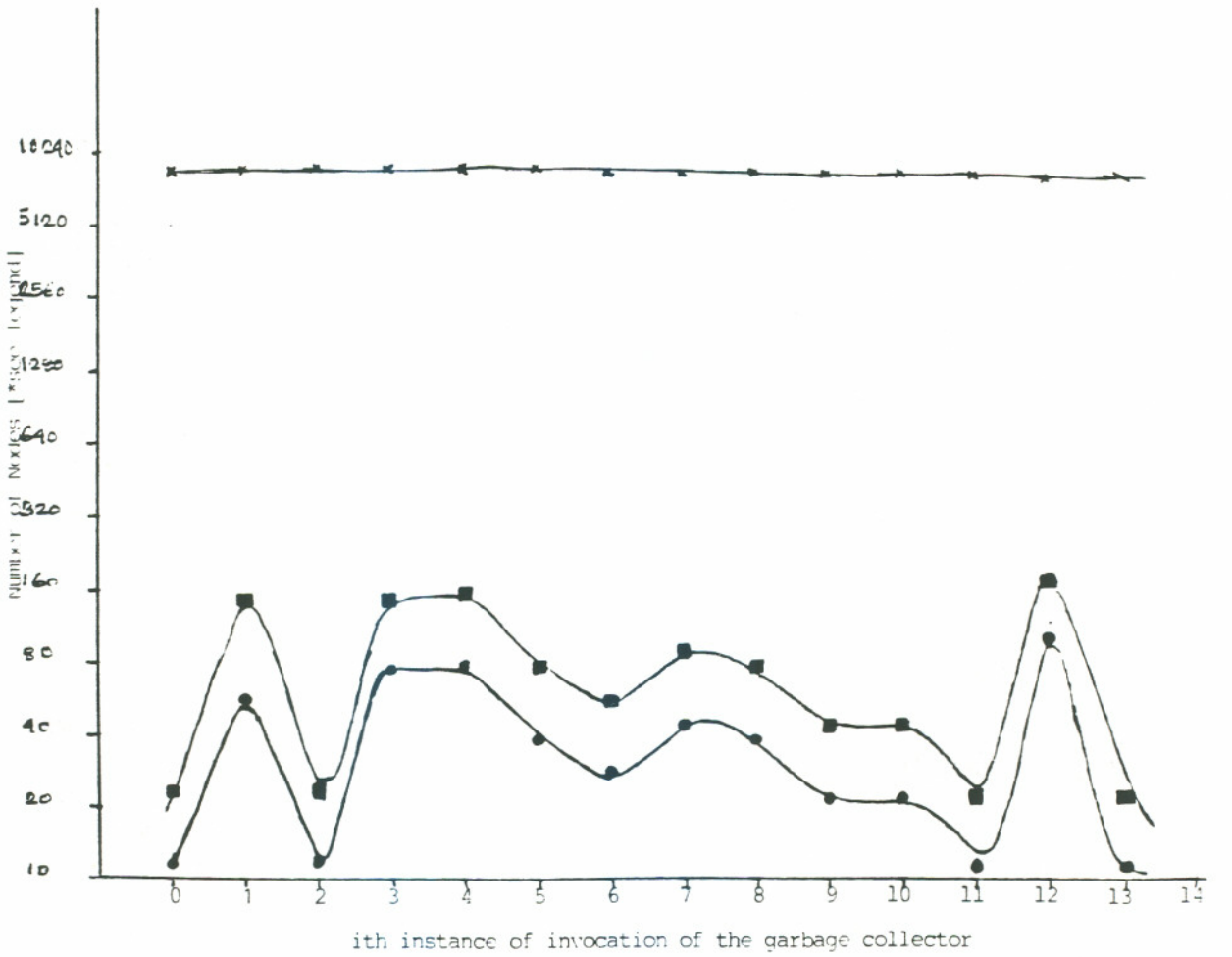


LEGEND: Top Graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected.

Seed = 7774700
Size of Garbage-Can = s = 4

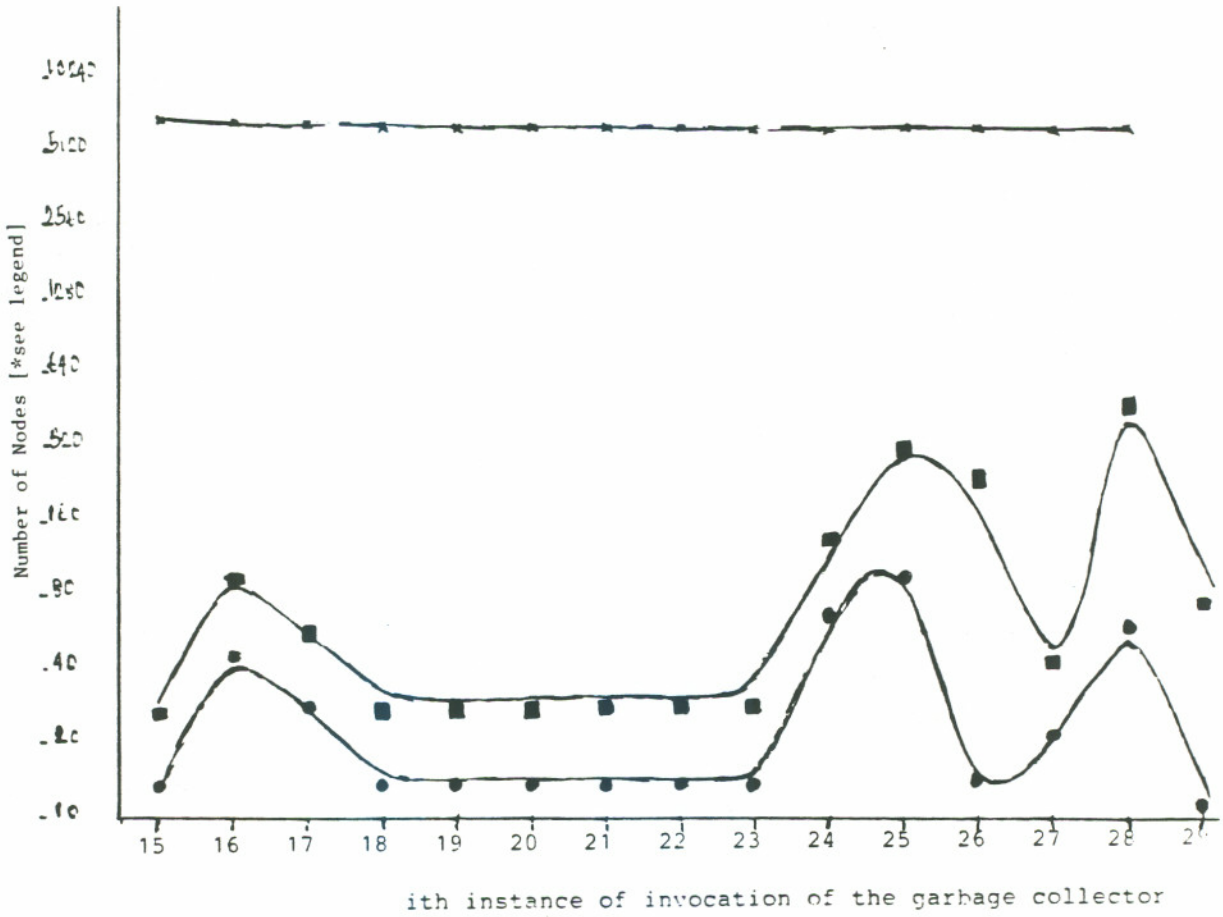


LEGEND: Top Graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected.

Seed = 7774700
Size of Garbage-Can = s = 4



LEGEND: Top Graph---The number of nodes traversed by conventional marking algorithm.

Middle Graph---The number of nodes traversed by the given coloring algorithm.

Lower Graph---The number of garbage nodes to be collected.

Seed = 7774700
Size of Garbage-Can = s = 4