Programming Techniques
S. L. Graham, R. L. Rivest Editors

A Time- and Space-Efficient Garbage Compaction Algorithm

F. Lockwood Morris
Syracuse University

Given an area of storage containing scattered, marked nodes of differing sizes, one may wish to rearrange them into a compact mass at one end of the area while revising all pointers to marked nodes to show their new locations. An algorithm is described here which accomplishes this task in linear time relative to the size of the storage area, and in a space of the order of one bit for each pointer. The algorithm operates by reversibly encoding the situation (that a collection of locations point to a single location) by a linear list, emanating from the pointed-to location, passing through the pointing locations, and terminating with the pointed-to location's transplanted contents.

Key Words and Phrases: Garbage collection, compaction, compactification, storage reclamation, storage allocation, record structures, relocation, list processing, free storage, pointers, data structures
CR Categories: 4.34, 4.49, 5.32

The Problem and Existing Solutions

Given an area of storage divided into *nodes* of which some are *marked* so as to be preserved, and which may be "pointed to" by addresses (*pointers*) stored in nodes or in other known locations outside the area, the *garbage compaction* problem is as follows: Rearrange the storage area so as to bring all marked nodes to contiguous positions towards one end, leaving the remainder of the area as a single block of "garbage." Since, in the process of compaction, the "points to" relation is to be preserved, pointers to nodes which are moved have to be *updated*—that is, revised—to give the new locations of the nodes to which they point—and here the difficulty arises: The only time it is natural to know where a given node is going to be moved is just when one is about to move it. This time will be determined by the distribution of marked nodes in the store and by the intended pattern of node movement. It cannot, to all appearances, be made to coincide with times of encountering all the arbitrarily distributed pointers to the given node.

Previous solutions [1–9 and 11] (see [10] for descriptive references to most of these solutions) rely on recording "forwarding addresses" for nodes at or near their original locations as their destinations are discovered. (The actual moving may or may not occur at this time.)

A summary of existing solutions follows. Each suffers from either (1) requiring a nontrivial amount of additional working storage, (2) taking time which either is worse than linear in the size of the storage area or is governed by the speed of secondary storage devices, or (3) placing restrictions on the allowed sizes of nodes.

(i) One may simply reserve a "forwarding address" field in each node, but this will be extravagant of space if the mean node size is small [6].

(ii) One may recognize that any block of contiguous marked nodes logically requires only a single "forwarding increment" to be recorded, and that at the end of each such block there will necessarily be a finite quantity of garbage, presumably large enough to store the increment. This method, however, requires a search to the end of its target block for the updating of each pointer, and therefore has a running time worse than linear in the sum of the storage size and the number of pointers [8, 11].

The preceding two schemes favor "planning to move"—i.e. recording forwarding addresses—followed by pointer updating, followed by moving. Alternatively, one may begin moving at once, and record each node's forwarding address in the space it formerly occupied, provided that the spot vacated by one node will not subsequently be required by another. This idea gives rise to the two following schemes:

(iii) Acquire an empty storage area temporarily and copy all marked nodes into it compactly. In practice, this method pays a time rather than a space penalty, either by using a paged virtual memory, or by explicitly writing the compacted nodes onto secondary storage and then reading them back [4, 7].

(iv) Arrange that each node moved will be written over what was initially garbage. To be sure of compacting completely in one step, one must require that all nodes be of the same size [6].

This research was partially supported by NSF Grant MCS75-22002.

Author's present address: School of Computer and Information Science, 313 Link Hall, Syracuse University, Syracuse, New York 13210.

Fig. 1.

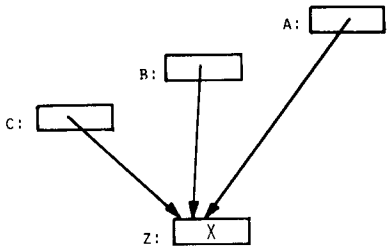


Fig. 2.

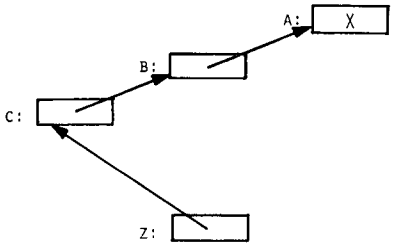
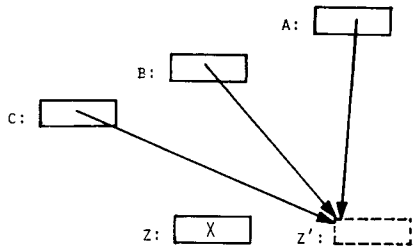


Fig. 3.



Development of the New Algorithm

The algorithm to be presented here operates in linear time, requires about one additional bit per pointer field for its own bookkeeping purposes, and allows nodes of any size greater than or equal to that of a single pointer. Its pattern of node movement will be that which naturally suggests itself for compaction of arbitrary-size nodes: "sliding," that is, movement of marked nodes to their new positions without alteration of their original linear order. (Note that the image which springs to mind—that of a bulldozer pushing an ever-growing mass of material in front of it—applies only to the garbage (the holes between the marked nodes). The nodes themselves are treated like potatoes in a potato race: Successive ones are fetched from ever more remote locations.) Sliding has the great virtue that the slid nodes hold together of themselves. Our compactor can be much simpler in that it may be entirely ignorant of the structure of nodes and regard storage as merely a succession of recognizable fields. Each field may or may not contain a pointer, and each independently (for all it knows) may be marked or not, subject to the constraint that any marked pointer lead to a field which is also marked. (In particular, no difficulty is caused by the existence of pointers to different fields of one node—a situation which arises in ECL[2] where nodes may physically incorporate subnodes.)

The algorithm manages updating by a reversible rearrangement of pointers, according to the following idea. Suppose locations *a*, *b*, and *c* all point to location *z*, and *z* has some contents *X* (Figure 1). Then by successive visits to *a*, *b*, and *c* we may again represent the situation without loss of information by constructing a list of locations which mean to point to *z*, emanating from *z*, and with the original contents of *z* saved at the end of the list (Figure 2). On a subsequent visit to *z*, at a time when the spot to which the field now at *z* will be moved is known, the status quo ante can be restored, but with updated values of the pointers (Figure 3), provided *X* recognizably is not a continuation of the list.

The restrictions which the representation of nodes and pointers must obey for these manipulations to be possible should be evident: We must be able to recognize a pointer; all pointer fields must be the same size and not too small (e.g. half words in a word-addressed machine) to be individually pointed at; and each pointer must claim a pointer-sized "target area" such that unequal pointers claim disjoint areas. All pointers to a node commonly point to the same end of it, and this last requirement becomes just the requirement that the smallest node be large enough to store a pointer. Moreover, when traversing a list, one must be able to distinguish the value at the end from the constituent links. This requires an additional bit of information for each of the locations involved (including *z*, even though it may not have contained a pointer originally).

The updating scheme just described appears likely to lead to chaos when applied to the entire storage structure, when one observes that the same location may temporarily be given an unnatural content for two different reasons—because it is pointed at and because it points. The way out of this difficulty lies in realizing, first, that if only all pointers pointed in the same direction (say from low-numbered locations to higher ones), then one could arrange that each location be finished with its role as a list head (with all pointers to it updated) before it needed to be considered in its role as a container of a pointer. Second, one must realize that one can in effect achieve this desirable state of affairs by performing the whole process twice, each time ignoring the pointers in the "wrong" direction. (Pointers from locations to themselves are an annoying special case, but are easily handled as such.) Updating, then, can be performed in two end-to-end sweeps of the storage area; if the first is made in the direction of compaction, then the actual movement of nodes (which necessarily progresses in the opposite direction) can be combined with the second updating sweep.

Implementation

In the version of the compaction algorithm which follows, it is supposed that the area to be compacted is a segment $M[l]$ through $M[h]$ of an integer array M, each array element being one field, and that a Boolean array *marked* contains in positions l through h a mark bit for each corresponding word of M. For convenience, all pointers into the node storage area from outside are taken to lie in a disjoint segment of M, in locations *sl* through *sh*. All values in M which themselves lie in the range l through h are understood to be pointers.

It is convenient to allow for the bookkeeping demands of the algorithm by the understanding that there is a constant *shift* such that the range of values l + *shift* through h + *shift* and *sl* + *shift* through *sh* + *shift* are guaranteed not to occur in M. The effect of this is that with every pointer there is an extra bit which is at our disposal, and which we may set and clear by adding and subtracting *shift*. Alternatively, one could provide another Boolean array, but the *shift* device emphasizes that each extra bit belongs to a pointer and moves with it; values which are not pointers, although they must be distinguishable as such, do not need the bit.

The marking routine is supposed to have done its work so as to establish the truth of the assertion

$(\forall k)((sl \leqslant k \leqslant sh$ **or** $(l \leqslant k \leqslant h$ **and** *marked* $[k]))$ **and**
$l \leqslant M[k] \leqslant h)$ **implies** *marked* $[M[k]]$.

The marking routine is also expected to have computed the quantity g of garbage, i.e. the number of indices $l \leqslant i \leqslant h$ for which *marked* $[i]$ is false. Compaction is to be towards h.

The notation used here is Algol 60 with the following exceptions for easier reading: 1) Each **for** clause is taken to declare its own controlled variable implicitly, with scope limited to the body of the **for** statement. A **for** clause which specifies only a single execution of the body (no **step** or **until**) thus provides a convenient form of initialized declaration. 2) The **while** ... **do** ... form of iterative statement is employed. 3) Conjunctions of inequalities are telescoped, e.g. $a < b \leqslant c$.

```
procedure compact (M, marked, l, h, sl, sh, shift, g);
  integer array M;
  Boolean array marked;
  integer value l, h, sl, sh, shift, g;
  begin integer n; comment new location counter;
    for i := sl step 1 until sh do
      for j := M[i] do
        if l ≤ j ≤ h then begin M[i] := M[j]; M[j] := i + shift end;
    n := l + g; comment prepare for sweep updating upwards pointers
          and those from outside;
    for i := l step 1 until h do if marked [i] then
    begin
      while l + shift ≤ M[i] ≤ h + shift or sl + shift ≤ M[i] ≤ sh +
          shift do
        for j := M[i] − shift do
          begin M[i] := M[j]; M[j] := n end;
      for j := M[i] do
          if i < j ≤ h then begin M[i] := M[j]; M[j] := i + shift end;
      n := n + 1
    end;
    n := h; comment prepare for sweep updating downwards pointers
          and compacting;
    for i := h step −1 until l do if marked [i] then
      begin
        while l + shift ≤ M[i] ≤ h + shift or sl + shift ≤ M[i] ≤ sh +
            shift do
          for j := M[i] − shift do
            begin M[i] := M[j]; M[j] := n end;
        for j := M[i] do
          if l ≤ j < i then begin M[i] := M [j]; M[j] := n + shift end
          else if j = i then M[i] := n;
        M[n] := M[i];
        n := n − 1
      end
  end compact
```

To provide the raw material for a proof of correctness of this procedure, we may state an invariant for each of the three outer loops which gives the current representation of a "typical fact" about the original contents of M, of the form $M[k] = m$ where k is between l and h and *marked* $[k]$ is **true**, or k is between *sl* and *sh*. Each invariant is true immediately after every assignment to i by its **for** clause, including assignment of the final excessive value with which the loop body is not executed.

For any q between l and h with *marked* $[q]$ **true**, let q' be the compacted location of q, i.e.

$$q' = q + \sum_{r=q+1}^{h} \textbf{if } marked\ [r] \textbf{ then } 0 \textbf{ else } 1.$$

(Note that both the second and third loops maintain $n = i'$.)

For the loop from *sl* up to *sh*: (i) If $sl \leqslant k < i$ and $l \leqslant m \leqslant h$, then for some $p > 0$ there exist $k_1, \ldots, k_p$ with $M[m] = k_1 + shift$, $M[k_1] = k_2 + shift, \ldots, M[k_{p-1}] = k_p + shift$, $k_p = k$. (ii) If $l \leqslant k \leqslant h$, then for some $p \geqslant 0$ there exist $k_1, \ldots, k_p$ with $M[k] = k_1 + shift, \ldots, M[k_p] = m$. (iii) Otherwise, $M[k] = m$. Initially, with $i = sl$, every word from $M[l]$ through $M[h]$ falls under case (ii) with $p = 0$; every word from $M[sl]$ through $M[sh]$ falls under (iii). The effect of the loop body is to transfer $M[i]$ from case (iii) to case (i) if it contains a pointer.

For the loop from l up to h: (iv) If $sl \leqslant k \leqslant sh$ and $l \leqslant m < i$, or $l \leqslant k < m < i$, then $M[k] = m'$. (v) If $sl \leqslant k \leqslant sh$ and $i \leqslant m \leqslant h$, or $l \leqslant k < i \leqslant m \leqslant h$, then for some $p > 0$ there exist $k_1, \ldots, k_p$ with $M[m] = k_1 + shift, \ldots, k_p = k$. (vi) If $i \leqslant k \leqslant h$, then for some $p \geqslant 0$ there exist $k_1, \ldots, k_p$ with $M[k] = k_1 + shift, \ldots, M[k_p] = m$. (vii) Otherwise, $M[k] = m$. Initially, with $i = l$, every word from $M[l]$ through $M[h]$ falls under case (vi); every word from $M[sl]$ through $M[sh]$ falls under (v) or (vii). Each execution of the loop body first (via the **while** statement) removes the word $M[i]$ from the domain of case (vi) to that of case (vii), simultaneously bringing any words which fell under case (v) and originally contained i into the domain of case (iv). $M[i]$ having recovered its original value, is then placed if necessary under case (v). When finally $i = h + 1$, only (iv) and (vii) are possible.

For the loop from h down to l: (viii) If $sl \leqslant k \leqslant sh$ and $l \leqslant m \leqslant h$, or $i < k < m \leqslant h$, or $h \geqslant k \geqslant m > i$, then $M[$**if** $h \geqslant k > i$ **then** k' **else** $k] = m'$. (ix) If $h \geqslant k > i \geqslant$

$m \geq l$, then for some $p > 0$ there exist $k_1, \ldots, k_p$ with $M[m] = k_1 + shift, \ldots, k_p = k'$. (x) If $i \geq k \geq l$, then for some $p \geq 0$ there exist $k_1, \ldots, k_p$ with $M[k] = k_1 + shift, \ldots, M[k_p] =$ **if** $k < m \leq h$ **then** m' **else** m. (xi) Otherwise, $M[$**if** $h \geq k > i$ **then** k' **else** $k] = m$. Cases (viii)–(xi) here and the transitions of words between them are homologous with cases (iv)–(vii) for the previous loop. When at first $i = h$, the words from $M[sl]$ through $M[sh]$ fall under cases (viii) and (xi), those from $M[l]$ through $M[h]$ fall under case (x) with $p = 0$. When at last $i = l - 1$, only (viii) and (xi) are possible. In short,

$M[$**if** $l \leq k \leq h$ **then** k' **else** $k]$
$=$ **if** $l \leq m \leq h$ **then** m' **else** m.

The running time of the algorithm would be self-evidently linear in the size of M, were it not for the embedded **while** loops. One quickly observes, however, that the **while**'s cannot be gone around more times in total than there are words in M, because each **while** iteration "unshifts" a pointer which can only have been shifted during a previous iteration of the enclosing **for** loop, or of the initial loop from sl up to sh.

Remarks

In applications it is all too likely that the compactor will have to be adapted to decipher the node structure of storage, for any of the following reasons: (1) The marking routine may record only one mark bit per node, (2) pointer fields may not be recognizable as such by their contents, and (3) the possible pointer fields may not recur at regular intervals. This being so, it is desirable that the ability to read off the nodes in a linear sweep of storage should be demanded in only one direction. We can meet this restriction by harking back to the observation that the nodes to be preserved fall into solid blocks separated by holes of positive size. The first sweep, which must of course be made in the legible direction (we suppose this is still l-to-h) can leave a pointer at the h end of each hole which links the blocks together in the h-to-l direction. The second sweep can then proceed along this chain, processing within each block in the l-to-h direction; it suffices to expand the formerly isolated special case $M[i] = i$ to take in all pointers with $M[i] \leq i$ but in the same block as i. These can all be updated directly by the addition of the distance by which their block is to be moved. It is probably best under these circumstances to split off moving the nodes into a third sweep of its own, since either overall or within each block moving must disagree in direction with the second updating sweep.

The algorithm can easily be modified to compact each of a collection of disjoint but mutually pointing storage areas by considering these areas as lying in an arbitrary linear order and treating them for the purpose of updating as one area.

Finally, it may be noted that there are applications in which nodes are created at the h end of the single block of known garbage and are never altered, though they may be abandoned. Since any pointer created must be to an already existing node, all will run in the l-to-h direction, and in this case only one updating sweep is necessary.

Received February 1977; revised August 1977

References

1. Cheney, C.J. A nonrecursive list compacting algorithm. *Comm. ACM 13,* 11 (Nov. 1970), 677–678.
2. Conrad, W.R. A compactifying garbage collector for ECL's non-homogeneous heap. Tech. Rep. 2–74, Ctr. for Res. in Compt. Tech., Harvard U., Cambridge, Mass., Feb. 1974.
3. Fenichel, R.R., and Yochelson, J.C. A LISP garbage collector for virtual-memory computer systems. *Comm. ACM 12,* 11 (Nov. 1969), 611–612.
4. Hansen, W.J. Compact list representation: Definition, garbage collection and system implementation. *Comm. ACM 12,* 9 (Sept. 1969), 499–507.
5. Hart, T.P., and Evans, T.G. Notes on implementing LISP for the M-460 computer. In *The Programming Language LISP: Its Operation and Applications,* Berkeley, E. C. and Bobrow, D. G., Eds., Information International, Cambridge, Mass., 1964, pp. 191–203.
6. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968, p. 421, Ex. 2.3.5.9 and p. 454, Ex. 2.5.33.
7. Minsky, M.L. A LISP garbage collector using serial secondary storage. M.I.T. Artif. Intell. Memo No. 58 (rev.), M.I.T., Cambridge, Mass., Dec. 1963.
8. Reynolds, J.C. Description of garbage collection in the COGENT programming system. Private communication.
9. Saunders, R.A. The LISP system for the Q-32 computer. In *The Programming Language LISP: Its Operation and Applications,* Berkeley, E. C. and Bobrow, D. G., Eds., Information International, Cambridge, Mass., 1964, pp. 220–231.
10. Steele, G.L. Multiprocessing Compactifying Garbage Collection. *Comm. ACM 18,* 9 (Sept. 1975), 495–508.
11. Wegbreit, B. A generalized compactifying garbage collector. *Computer J. 15,* 3 (Aug. 1972), 204–208.