

Syracuse University

SURFACE

Electrical Engineering and Computer Science -
Technical Reports

College of Engineering and Computer Science

2-1977

A Time- and Space-Efficient Garbage Compaction Algorithm

F. Lockwood Morris

Syracuse University, lockwood@ecs.syr.edu

Follow this and additional works at: https://surface.syr.edu/eecs_techreports



Part of the [Computer Sciences Commons](#)

Recommended Citation

Morris, F. Lockwood, "A Time- and Space-Efficient Garbage Compaction Algorithm" (1977). *Electrical Engineering and Computer Science - Technical Reports*. 43.

https://surface.syr.edu/eecs_techreports/43

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

A TIME- AND SPACE-EFFICIENT
GARBAGE COMPACTION ALGORITHM

F. Lockwood Morris

February 1977



SCHOOL OF COMPUTER
AND INFORMATION SCIENCE
SYRACUSE UNIVERSITY

A Time- and Space-Efficient Garbage Compaction Algorithm

F. Lockwood Morris
Syracuse University

Abstract: Given an area of storage containing scattered marked nodes, one may wish to rearrange them into a compact mass at one end of the area, meanwhile revising all pointers to marked nodes to show their new locations. An algorithm is here described which accomplishes this task in linear time relative to the size of the storage area, and in 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

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.

A Time- and Space-Efficient Garbage Compaction Algorithm

The Problem and Existing Solutions

Given an area of storage divided into *nodes*, which are some of them *marked* 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 to 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". In the process of compaction, since 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 when 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, and cannot to all appearances be made to coincide with times of encountering all the arbitrarily distributed pointers to the given node.

Previous solutions [1, 2, 3, 4, 5, 6, 7, 8, 9, 11] - see Steele's article [10] for descriptive references to most of these - 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 requiring a non-trivial amount of additional working storage, or from 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 from 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 mean node size is small.

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

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 following two schemes:

(iii) Temporarily acquire an empty storage area, 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.

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

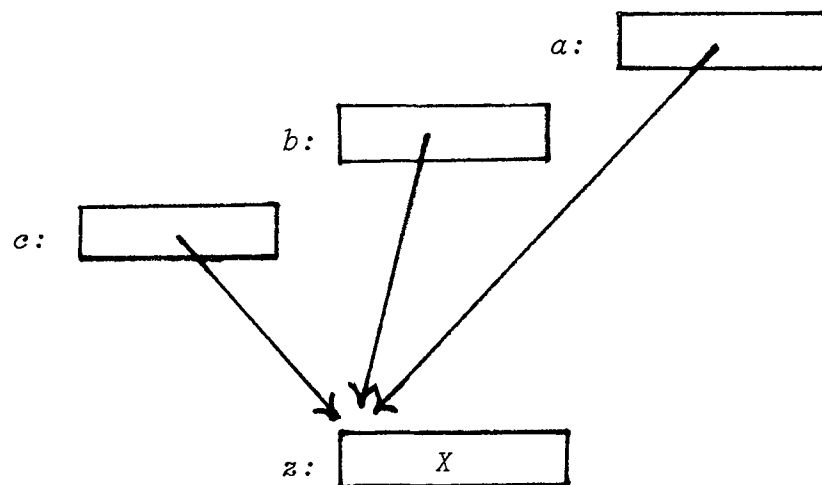
Development of the New Algorithm

The algorithm to be presented here operates in linear time and requires about one additional bit per pointer field for its own bookkeeping purposes. 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 alternation 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 experience of the nodes being compacted is like that of the potatoes in a potato race, successive ones being fetched from ever more remote locations.) Sliding has the great virtue that the slid nodes hold together of themselves; our compactor can be so much the simpler in that it may be entirely ignorant of the structure of nodes, and

regard storage as merely a succession of recognizable fields, each containing or not containing a pointer, and each independently (for all it knows) marked or not, subject to the constraint that any marked pointer lead to a field which is also marked.

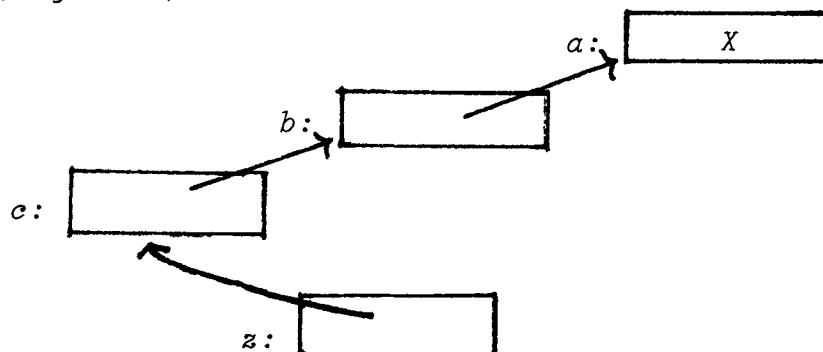
The algorithm manages updating by a reversible rearrangement of pointers, according to the following idea. Suppose locations a , b , c all point to location z , and z has some contents X (Figure 1).

Fig. 1



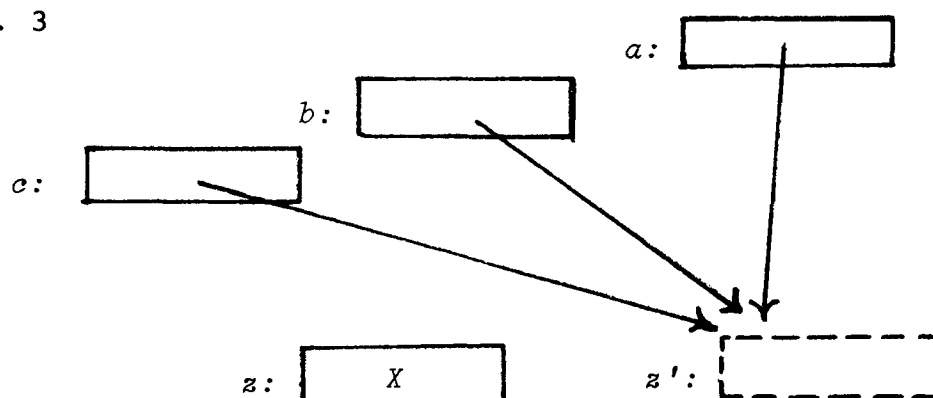
Then by successive visits to a , b , and c we may re-represent the situation without loss of information by constructing a list of locations which mean to point to z , emanating from z , and with the original contents of z saved at the end of the list (Figure 2).

Fig. 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 is recognizably not a continuation of the list.

Fig. 3



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. In the common case, all pointers to a node point to the same end of it, and this last requirement becomes just 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, as soon as 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) one could arrange that each location had done with its role as a list head, with all pointers to it updated, before it needed to be considered in its role as container of a pointer; second, 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 , 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 not by the allocation of an additional Boolean array, but by the understanding that there is a constant *shift* such that the range of values $l + \text{shift}$ through $h + \text{shift}$ and $sl + \text{shift}$ through $sh + \text{shift}$ are guaranteed not to occur in M . The marking routine is supposed to have done its work so as to establish the truth of the assertion

$$(\forall k) ((sl \leq k \leq sh \text{ or } (l \leq k \leq h \text{ and } \text{marked } [k])) \text{ and } l \leq M[k] \leq h) \text{ implies } \text{marked } [M[k]].$$

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

The notation used here is meant for good Algol 60 with the following exceptions in favor of readability:

- 1) Each for clause is taken to declare its own controlled variable implicitly, with scope limited to the body of the for statement.
- 2) The while ... do ... form of iterative statement is employed.
- 3) Conjunctions of inequalities are telescoped, e.g. $a < b \leq c$.

```

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  $l \leq j \leq h$  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  $l+shift \leq M[i] \leq h+shift$  or  $sl+shift \leq M[i] \leq sh+shift$  do
        for j:=M[i]-shift do
          begin M[i]:=M[j]; M[j]:=n end;
        for j:=M[i] do
          if  $i < j \leq h$  then begin M[i]:=M[j]; M[j]:=i+shift end;
        n:=n+1
      end;

  n:=h; comment prepare for sweep updating remaining
    pointers and compacting;

```

```

for  $i:=h$  step -1 until  $l$  do if  $\text{marked}[i]$  then
  begin
    while  $l+\text{shift} \leq M[i] \leq h+\text{shift}$  or  $sl+\text{shift} \leq M[i] \leq sh+\text{shift}$  do
      for  $j:=M[i]-\text{shift}$  do
        begin  $M[i]:=M[j]$ ;  $M[j]:=n$  end;
      for  $j:=M[i]$  do
        if  $l \leq j < i$  then begin  $M[i]:=M[j]$ ;  $M[j]:=n+\text{shift}$  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 two main 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 $\text{marked}[i]$ 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 $\text{marked}[q]$ true, let q' be the compacted location of q , i.e.

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

(Note that both loops maintain $n=i'$.)

For the loop from l up to h :

- (i) If $sl \leq k \leq sh$ and $l \leq m < i$, or $l \leq k < m < i$, then $M[k] = m'$.
- (ii) If $sl \leq k \leq sh$ and $i \leq m \leq h$, or $l \leq k < i \leq m \leq h$, then for some $p > 0$ there exist k_1, \dots, k_p with $M[m] = k_1 + shift$, $M[k_1] = k_2 + shift, \dots, k_p = k$.
- (iii) If $i \leq k \leq h$, then for some $p \geq 0$ there exist k_1, \dots, k_p with $M[k] = k_1 + shift, \dots, M[k_p] = m$.
- (iv) Otherwise, $M[k] = m$.

Initially, with $i = l$, every word from $M[l]$ through $M[h]$ falls under case (iii) with $p = 0$; every word from $M[sl]$ through $M[sh]$ under (ii) or (iv). Each execution of the loop body first (*via* the while statement) removes the word $M[i]$ from the domain of case (iii) to that of case (iv), simultaneously bringing any words which fell under case (ii) and originally contained i into the domain of case (i). $M[i]$ having recovered its original value, it is then if necessary placed under case (ii). When finally $i = h + 1$, only (i) and (iv) are possible.

The preparatory loop from sl to sh has a similar but simpler invariant: case (ii) applies for $sl \leq k < i$ with $l \leq m \leq k$, case (iii) for $l \leq k \leq h$, and case (iv) otherwise.

For the loop from h down to l :

- (v) If $sl \leq k \leq sh$ and $l \leq m \leq h$, or $l \leq k < m \leq h$, or $h \geq k \geq m > i$, then $M[\text{if } h \geq k > i \text{ then } k' \text{ else } k] = m'$.
- (vi) If $h \geq k > i \geq m \geq l$, then for some $p > 0$ there exist k_1, \dots, k_p with $M[m] = k_1 + shift, \dots, k_p = k'$.
- (vii) If $i \geq k \geq l$, then for some $p \geq 0$ there exist k_1, \dots, k_p with $M[k] = k_1 + shift, \dots, M[k_p] = \text{if } k < m \leq h \text{ then } m' \text{ else } m$.

(viii) Otherwise, $M[\text{if } h \geq k > i \text{ then } k' \text{ else } k] = m$.

Cases (v)-(viii) here and the transitions of words between them are homologous with cases (i)-(iv) for the previous loop. When finally $i = l-1$, we have

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

The running time of the algorithm would be self-evidently linear in the size of M , but for the embedded while loops. One quickly observes, however, that the while's cannot be gone round in total more times 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 reasons that the marking routine may record only one mark bit per node, that pointer fields may not be recognizable as such by their contents, or that 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 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 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 it must disagree in direction with the second updating sweep.

Our algorithm can easily be modified to compact each of a collection of disjoint but mutually pointing storage areas, by considering them to lie in an arbitrary linear order and treating them for purposes 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.

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, Center for Research in Computing Technology, 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. *The Programming Language LISP: Its Operation and Applications*. Berkeley and Bobrow (Eds.), Information International, Inc., Cambridge, Mass., 1964, 191-203.
6. Knuth, D. E. *The Art of Computer Programming Vol. 1 Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968, p. 421 Exercise 2.3.5.9 and p. 454 Exercise 2.5.33.
7. Minsky, M. L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo No. 58 (revised), MIT 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. *The Programming Language LISP: Its Operation and Applications*. Berkeley and Bobrow (Eds.), Information International, Inc., Cambridge, Mass., 1964, 220-231.
10. Steele, G. L. Multiprocessing Compactifying Garbage Collection, *Comm. ACM* 18, 9 (Sept. 1975), 495-508.
11. Wegbreit, B. A generalised compactifying garbage collector. *Computer J.* 15, 3 (Aug. 1972), 204-208.