Pathological Cases for a Class of Reachability-Based Garbage Collectors

MATTHEW SOTOUDEH, Stanford University, USA

Although existing garbage collectors (GCs) perform extremely well on typical programs, there still exist pathological programs for which modern GCs significantly degrade performance. This observation begs the question: might there exist a 'holy grail' GC algorithm, as yet undiscovered, guaranteeing both constant-length pause times and that memory is collected promptly upon becoming unreachable? For decades, researchers have understood that such a GC is *not* always possible, i.e., some pathological behavior is unavoidable when the program can make heap cycles and operates near the memory limit, regardless of the GC algorithm used. However, this understanding has until now been only informal, lacking a rigorous formal proof.

This paper complements that informal understanding with a rigorous proof, showing with mathematical certainty that *every* GC algorithm that can implement a realistic mutator-observer interface has some pathological program that forces it to either introduce a long GC pause into program execution or reject an allocation even though there is available space. Hence, language designers must either accept these pathological scenarios and design heuristic approaches that minimize their impact (e.g., generational collection), or restrict programs and environments to a strict subset of the behaviors allowed by our mutator-observer–style interface (e.g., by enforcing a type system that disallows cycles or overprovisioning memory).

We do not expect this paper to have any effect on garbage collection practice. Instead, it provides the first mathematically rigorous answers to these interesting questions about the limits of garbage collection. We do so via rigorous reductions between GC and the dynamic graph connectivity problem in complexity theory, so future algorithms and lower bounds from either community transfer to the other via our reductions.

We end by describing how to adapt techniques from the graph data structures community to build a garbage collector making worst-case guarantees that improve performance on our motivating, pathologically memory-constrained scenarios, but in practice find too much overhead to recommend for typical use.

CCS Concepts: • **Software and its engineering → Garbage collection**.

Additional Key Words and Phrases: Garbage Collection, Programming Languages, Complexity

ACM Reference Format:
Matthew Sotoudeh. 2025. Pathological Cases for a Class of Reachability-Based Garbage Collectors. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 96 (April 2025), 28 pages. https://doi.org/10.1145/3720430

1 Introduction

Managing limited memory resources is a fundamental problem in programming language design. Garbage collection (GC) is a user-friendly approach where the language runtime automatically deallocates memory regions once they are no longer reachable from the local and global variables [Jones et al. 2011]. Concerns about predictability and pause times have led many low-level languages to adopt manual memory management, where the programmer explicitly tells the runtime when to release resources [Wolfe 2017]. Other languages use type systems that allow the compiler to predict statically where to release memory resources at the cost of restricting program expressiveness [Coblenz et al. 2022]. This paper explains in a formal way why these tradeoffs are necessary, by proving hard limits on the asymptotic worst-case performance of a wide class of GCs.

Author's Contact Information: Matthew Sotoudeh, Stanford University, Stanford, USA, sotoudeh@stanford.edu.

ACM 2475-1421/2025/4-ART96
https://doi.org/10.1145/3720430

We first give a formal model characterizing precisely the class of garbage collectors our results apply to. In particular, we address only those GC algorithms that can be used to implement a mutator-observer interface where the GC algorithm reads a stream of pointer updates from the program and then must add regions to a free list once they become unreachable. In such a setting, one key issue is how long it takes for the GC to add a region to the free list after it becomes unreachable: we call this the *delay*. Many real-time settings, e.g., medical devices and avionics, have critical timing constraints. In those settings, it can be catastrophic for the program to be forced to wait for a significant period of time on the GC to finish collecting a region before a new allocation can be made.

Until a memory limit is reached, state-of-the-art real-time garbage collection algorithms can make the following guarantees (see, e.g., Section 4.2 of Bacon et al. [2003]):

(1) **Constant Pause Times:** The GC only slows each program instruction down by a constant amount.
(2) **Linear Collection Delay:** Suppose at some program point there are n reachable memory regions, and then a program operation makes one of those regions unreachable. Then, the GC guarantees that region will be collected within $O(n)$ program operations.

The fundamental question of this paper is whether the $O(n)$ collection delay can be reduced without increasing pause times. For naïve tracing GC algorithms, which are designed around a traversal of the graph, the answer is intuitively no: no regions can be safely collected until the entire traversal, which may have to visit $O(n)$ nodes, completes. While the story is complicated by the use of incremental tracing, among GC researchers, there is an informal understanding that the $O(n)$ collection delay is unavoidable in some pathological scenarios, where the program may make heap cycles and operates close to the memory limit. However, until this paper, there was no formal proof ruling out the existence of fundamentally more-clever algorithms yet to be discovered that improve collection delay without worsening pause times.

1.1 Problems with Delay

Before explaining the theoretical results of our paper, it is helpful to motivate our study by describing one application-level issue (GC thrashing) caused by the existence of collection delay. This issue is well-known among both researchers and practitioners [Bloch 2017; Christian and Marks 2021; Nguyen et al. 2016; Venners 1998]; we repeat it here merely to keep the paper self-contained. A more in-depth explanation is provided in Section 2, and a separate issue compounding on the already-problematic language feature of finalization is described for interested readers in Appendix J.

Section 2.1 describes a program operating very close to the memory limit. It first allocates a large amount of memory that stays reachable through the entire program execution, pinning its logical memory usage near the limit. It then repeatedly makes a single region unreachable before requesting a new allocation in a loop. Because of collection delay, modern GCs do not guarantee that the region just made unreachable can be actually collected in time to be reused for the new allocation. Hence, to stay under the memory limit, the GC is forced to complete a full collection cycle on each iteration, introducing linear-length pause times that would be catastrophic for critical real-time applications. In our example, the GC ends up thrashing and slows down end-to-end execution time by 70× compared to a manually managed version[1].

[1]In practice, real-time systems based on collectors like that of Bacon et al. [2003] work around this issue by overprovisioning memory so the limit is never reached. Overprovisioning incurs additional costs and is difficult to apply in settings with dynamically changing memory constraints, e.g., with multiple interacting processes.

1.2 Impossibility Result

It is tempting to hope that the issues described above and in Section 2 could be fixed once-and-for-all with a more complicated collector. **The core result of this paper is an impossibility theorem implying it is fundamentally impossible to avoid such issues (Corollary 4.5).** For every garbage collector (defined formally in Section 3.1) there is a program where the GC must either (i) introduce a superlogarithmic pause during some program operation, or (ii) delay collecting a region for a superlogarithmic number of program operations. If a long pause can be introduced, the collector is insufficient for use in real-time settings (e.g., medical devices) where timing is critical. On the other hand, if collection delay is introduced, the collector is insufficient for applications that either (i) operate close to the memory limit and so must be able to reuse memory regions quickly; or (ii) rely on prompt finalization for actions like unlocking.

Notably, GC researchers have understood this fact for decades, but only at an informal level. To the best of our knowledge, ours is the first rigorous impossibility result concerning garbage collection. The result follows from a novel connection to the well-studied problem of *dynamic connectivity* in the graph algorithms community. Unfortunately, that connection goes both ways: although we expect the lower bound can be improved (e.g., from superlogarithmic to linear), this is exactly as difficult as solving a longstanding open problem in graph algorithms.

1.3 Limitations of Results

It is crucial to note a number of limitations to our results. First, they only imply lower bounds for a GC algorithm insofar as the GC algorithm can be used to implement our mutator-observer GCDS interface in Section 3.1. Hence, they say nothing directly about, e.g., moving collectors. However, it is sometimes possible to modify such GC algorithms to fit our GCDS interface without significant effect on asymptotic time. For example, Baker [1992] adapts the moving collector in Jr. [1978] to a nonmoving version that can be used to implement our GCDS interface. In those cases, our bounds apply only to the modified algorithm, and additional analysis must be done to understand to what extent (if at all) the results imply anything about the unmodified algorithm.

Second, our results address only the problem of collecting regions no longer reachable by pointers from a root set. We say nothing about GC approaches using different approximations to liveness.

Finally, they imply merely that there exists *at least one* pathological sequence of pointer operations where the GC algorithm performs poorly, but they say nothing about the performance of the algorithm in typical cases. In particular, the pathological behavior we prove exists will involve operating close to the memory limit and the existence of potential heap cycles. Hence, our results are sidestepped when these pathological cases are ruled out by restrictive type systems or resource overprovisioning, and they say nothing about what can be achieved for the typical case by well-designed heuristics like generational collection.

1.4 Implications of Results

Our motivation is theoretical, to understand with mathematical certainty the limits of GC. After nearly 60 years of research [Gelernter et al. 1960; Newell and Shaw 1957], there still exist pathological programs where even the best GCs significantly degrade performance (Section 2). This paper provides a precise, formal explanation for the extent to which such pathological cases are unavoidable, even by GC algorithms not yet discovered.

For GC researchers, our results provide precise and formal confirmation of the informal understanding that searching for GC algorithms that avoid all such pathological cases in the worst case is futile, hence motivating heuristics like generational collection. They also provide a helpful sanity check on claims; in Section 5.3 we describe how one of our lower bounds would have helped discover a known error in the cyclic reference counting literature.

Our analysis has no direct implication for GC users, beyond slightly clarifying the worst-case tradeoffs that will come with the use of automated GC. In the very long term, we are optimistic that the algorithm described in Section 7 might spur research ideas that lead to better garbage collectors, but our evaluation of the algorithm as presented in this paper indicates that it is only beneficial in very extreme pathological scenarios, hence not recommended for typical use.

1.5 Contributions and Outline

The major contributions of this paper are as follows:

(1) Definition of the *Garbage Collection Data Structure* which formalizes the mutator–observer interface of most nonmoving garbage collection schemes (Section 3).
(2) A novel lower bound showing fundamental tradeoffs between worst-case pause time and collection delay (Section 4). It implies that every GC fitting the GCDS interface is susceptible to pathological cases similar to the ones demonstrated in Section 2.1.
(3) A second lower bound showing that reference counting cannot be extended to handle cycles without either significantly increasing the worst-case pause times or introducing significant collection delay, even on programs making only acyclic heaps (Section 5).
(4) A GC guaranteeing $O(1)$ collection delay in all settings and logarithmic pause times while the heap is acyclic (Section 7). While it introduces too much overhead to suggest as a general-purpose GC, it addresses interesting and long-standing theoretical questions regarding the extent to which reference counting can be extended to handle cycles.

Section 6 ties the technical results back to the programming languages context, Section 8 discusses limitations and future work, Section 9 discusses related work, and Section 10 concludes the paper.

2 Motivating Example: What's Wrong with Delay?

We motivate the question of whether garbage collectors with constant pause times and collection delay exist using a well-known GC issue, thrashing [Nguyen et al. 2016]. Appendix A has full code and explains how to trigger similar behavior in GCs using generations, reference counting, etc. In addition to the thrashing problem, collection delay is known to cause deadlocks when combined with programs making overzealous use of finalizers. However, the use of finalizers is generally known to be bad practice, so we relegate an example of this to the appendix (Appendix J).

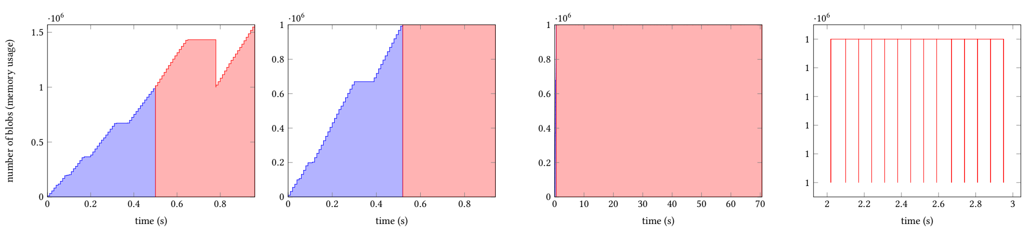
2.1 GC Thrashing and Linear Pause Times in the Memory-Constrained Setting

Real-time applications critically require constant-length worst-case pause times. Existing garbage collection schemes can guarantee constant-length pause times. But because they do not guarantee immediate collection, when a program operates close to the memory limit—common in, e.g., embedded settings where real-time guarantees are important—existing GCs can be forced to choose between breaking this guarantee and waiting significantly longer to complete a collection cycle before enough free memory can be found to continue execution, or refusing an allocation even though unreachable memory does exist.

Demonstrating this, consider the Lua program excerpted in Figure 1. The calls to `fetch_item` and `process_item` both allocate large memory buffers. The program attempts to limit overall memory usage via a small wrapper (not shown in the excerpt) that records the total number of uncollected allocations and refuses to allocate more if the limit is reached.

```lua
-- (excerpted implementations away of some functions, globals)
-- stage 1: fetch items into an array
-- each item has a large memory buffer associated with it
local items = {}
for i=1,N_ITEMS do items[i] = fetch_item() end
-- at this point, memory is near the limit
-- stage 2: compute a summary statistic
local summary = 0
-- process_item makes large, but temporary, allocations
-- frequent GC pauses needed to collect old temporary allocations
for i=1,N_ITEMS do summary = summary + process_item(items[i]) end
```

Fig. 1. Excerpt from the example program showing long pause times in the memory-constrained setting. Both `fetch_item` and `process_item` allocate large memory blobs using a wrapper that enforces a strict limit on the total (logical) allocation size. The limit is almost reached by the end of stage 1, and only the blobs allocated in stage 2 are temporary, so nearly every iteration of the loop in stage 2 needs to make a full GC pass to collect old temporary blobs before allocating a new one for that iteration. This program is sufficient to trigger GC thrashing in Lua. Variants of this program trigger similar linear-length pause times in collectors that use generational collection and reference counting.



(a) No limit (b) Manual memory (c) GC (d) GC zoomed in

Fig. 2. With and without memory pressure. The first stage is shown in blue, the second in red.

The application's first stage fills an array of objects. Each object contains a reference to a large memory blob, e.g., the contents of a file. These blobs will stay permanently reachable throughout the program execution. After the first stage we have nearly reached our memory limit.

The second stage computes a summary, e.g., the most frequent token in the file, from each item in the array. Computing the summary requires allocating another large, but temporary, memory buffer for each item. Hence, the memory limit gets reached quickly after the first few iterations of the second stage. Seeing this, the allocator must wait for the GC to finish collecting old temporary buffers from previous iterations before it can allocate space for the latest iteration of the second loop. This causes repeated waiting on the GC, often referred to as *GC thrashing*.

Logical memory usage is graphed against time in Figure 2. Without a memory limit (Figure 2a) the GC is never forced to run, so the application finishes quickly with only a small number of relatively short pauses. When a limit is enforced but the programmer manually marks unreachable memory for collection (Figure 2b), the limit is never reached so performance is similarly good.

However, when the memory limit is enforced and the GC is relied on to automatically reclaim unreachable regions (Figure 2c), we see a 70× slowdown. Nearly all of this time is spent in the second stage thrashing within the GC. Figure 2d shows a zoomed-in view of one second's worth of the second stage execution. The majority of time is spent with memory usage pinned at the maximum while the GC is running to free up new space for a new allocation. The actual application work only happens within the near instantaneous dips below that limit.

Note that all of the memory allocated during the first stage is still reachable, so the GC cannot collect more than a small amount at a time. Specifically, it collects temporary regions from calls to `process_item` since the last time it was run. Also note that the time taken by the GC is not time needed to actually free the memory, as the manual memory management performance is significantly better. Instead, the time is spent *locating what can be freed*: ensuring even a single region can be freed requires searching the *entire* heap to check that nothing else points to it.

This pathological behavior shows one downside of collection delay: when more memory is needed, there is no guarantee that the GC will be able to find it quickly even if it exists. Our impossibility result proves that this sort of scenario is impossible to avoid.

3 Preliminaries and the Garbage Collection Data Structure

To rigorously investigate claims about what is or is not possible for future garbage collectors, we must formalize what we mean by a garbage collector. This section formalizes the garbage collection problem as the garbage collection data structure (GCDS) and discusses nuances in defining the asymptotic running time of GCDS operations.

3.1 Garbage Collection Data Structure (GCDS)

We will model runtime pointer information as a directed multigraph, i.e., a set V of vertices (nodes) and a map $E : V \times V \longrightarrow \mathbb{N}_{\geq 0}$ indicating the number of edges from one node to another. Vertices represent memory regions and the edges represent pointers from one region to another (we will use "regions" and "nodes" interchangeably throughout). Because we are proving *lower* bounds, we are justified in ignoring the size of memory regions; the worst-case sequence of operations we will prove exists can be thought of as making equal-sized allocations.

A *GC heap* is a directed multigraph along with a distinguished vertex `root` $\in V$ with no incoming edges. The `root` vertex is meant to represent the local and global variables, i.e., it has outgoing edges to any regions that local and global variables have direct pointers to.

With this in mind, we can define a set of procedures that describe the desired interface between the programming language and the garbage collector. The interface was designed to capture the standard interface of nonmoving GCs for imperative code, i.e., where the GC sees the program as a mutator modifying pointers in a heap.

Definition 3.1. A *garbage collection data structure* (GCDS) is a data structure that (i) represents a GC heap, i.e., a directed multigraph with distinguished vertex `root`; (ii) stores a list `GCFreeList` of vertices not reachable from `root`; and (iii) supports the following procedures:

- `GCAllocate`() allocates a new vertex a and adds edge `root` $\longrightarrow a$.
- `GCInsert`($a \longrightarrow b$) adds edge $a \longrightarrow b$.
- `GCDelete`($a \longrightarrow b$) removes edge $a \longrightarrow b$.
- `GCStep`() requests the GCDS perform additional collection work (analogous to, e.g., `collectgarbage("step")` in Lua).

The GCDS may only add nodes that are unreachable from `root` to `GCFreeList`. While the user of the data structure may remove nodes from `GCFreeList`, the GCDS itself must never remove a node from `GCFreeList` (only add). The user of the data structure promises no edge ending in `root` is inserted and no query involves a node no longer reachable from `root`. We say each call made to one of these GCDS procedures is a *GCDS operation.*

A GCDS can be used to implement GC for an imperative language. When the program overwrites a pointer in region s that was pointing to region d_1 to now point to region d_2, the GCDS operations GCInsert($s \to d_2$) and GCDelete($s \to d_1$) are performed. GCStep() can be called after every program operation to perform some incremental GC work. When memory pressure is encountered, GCFreeList can be checked for allocation regions that can be reused by the application. If GCFreeList is empty and the application requires more memory, GCStep() can be called to request the collector spend more time searching for regions that may be safely freed. The extent to which an efficient GC implies the existence of an efficient GCDS is discussed in Appendix G.

Note that the collector does *not* need to ensure that all unreachable nodes are placed on GCFreeList immediately upon becoming unreachable, or even after a call to GCStep(). This allows for modelling concurrent tracing collectors that perform only a small portion of a sweep after each program operation. The frequency at which GCFreeList is updated to reflect newly unreachable nodes is captured by the *delay* metric below (Definition 3.2).

3.2 Defining Asymptotic Complexity of Garbage Collection

We consider two primary measures of GCDS efficiency. These definitions are somewhat informal for space reasons; more formal definitions and a discussion of alternative definitions that also work is given in Appendix H. We will assume here the GCDS is deterministic.

3.2.1 Worst-Case Delay. The focus of this paper is on *worst-case delay*, i.e., the number of GC operations that may need to be executed before an unreachable node is added to GCFreeList.

Definition 3.2. The *worst-case delay* $d(n)$ of a GCDS is the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes (i.e., making at most n calls to GCAllocate), if some operation makes a node u unreachable, u is added to GCFreeList after at most $d(n)$ more GCDS operations (including the operation that makes it unreachable).

3.2.2 Worst-Case Pause Times. The other quantity we are interested in is the *worst-case pause time*, i.e., the longest amount of time that any single GCDS operation might take. This corresponds directly to the extra pause time between program operations introduced by the GC. In a real-time application, guaranteeing $O(1)$ pause times can be critical.

Definition 3.3. The *worst-case pause time* $t(n)$ of a GCDS is the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes, each GCDS operation in the sequence takes time at most $t(n)$.

3.2.3 Fine-Grained Measures: Noncollecting and Acyclic Pause Times. We will eventually prove that, for every GCDS, there is a sequence of operations that forces it to introduce either a long pause time or a long collection delay. The reader may be concerned that the sequence of operations is somehow 'unfair,' e.g., that long pause times are only encountered because a large fraction of the heap needs to be collected quickly, or that densely connected structures are used, or that cyclic structures are used. To address these concerns we will need to introduce the *all-reachable*, *acyclic all-reachable*, and *sparse acyclic all-reachable* variants of the worst-case pause time definition from earlier. Each one captures the worst-case pause times introduced for a subset of possible program executions. Hence, lower bounds on these are stronger than lower bounds on $t(n)$, as they guarantee the existence of a hard sequence of operations involving progressively simpler classes of heaps.

Definition 3.4. The *all-reachable* worst-case pause time $t_{\text{AR}}(n)$ of a GCDS is the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes **and never making any node unreachable**, each operation in the sequence takes time at most $t_{\text{AR}}(n)$.

Definition 3.5. The *acyclic all-reachable* worst-case pause time $t_{\mathrm{A,AR}}(n)$ of a GCDS is the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes, **never making any node unreachable, and never forming a cycle in the heap**, each operation in the sequence takes time at most $t_{\mathrm{A,AR}}(n)$.

Definition 3.6. The *sparse acyclic all-reachable* worst-case pause time $t_{\mathrm{S,A,AR}}(n)$ of a GCDS is the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes, **never making any node unreachable, never forming a cycle in the heap, and never having more than $O(1)$ edges leaving any node**, each operation in the sequence takes time at most $t_{\mathrm{S,A,AR}}(n)$.

It is important to clearly note here that lower bounds on these restricted versions of $t(n)$ are *stronger* than lower bounding $t(n)$ itself.

Lemma 3.7. *Every GCDS has* $t_{\mathrm{S,A,AR}}(n) \leq t_{\mathrm{A,AR}}(n) \leq t_{\mathrm{AR}}(n) \leq t(n)$.

Proof. Every function in the sequence is defined as the max time over progressively larger subsets of possible executions. The claim follows because the max over a subset is smaller than that over the full set, i.e., when $S \subseteq T$ we know $\max(S) \leq \max(T)$. □

3.3 Examples of GCDS Implementations

Our theoretical results (Section 4 and Section 5) apply to any GC approach that can be adapted to implement the GCDS interface (Definition 3.1). It is therefore important to get a sense for how different GC approaches can be adapted to implement the GCDS interface. This section describes how to use both reference counting and mark-and-sweep to implement the GCDS interface.

In the below pseudocode, we make use of multisets to store the edges in the heap graph. We assume multisets are implemented such that the following can be done in $O(1)$ time:

(1) They can be converted into an iterator that only visits each element once; we call this operation $s.\mathrm{no\_multi}()$.
(2) The number of copies of an object in the multiset can be computed using $s.\mathrm{count}(o)$.

3.3.1 Eager Reference Counting as a GCDS. Figure 3 shows an implementation of the GCDS interface using the eager reference counting technique. The GCDS's internal data includes a dictionary mapping each node ID to a reference count, a multiset of outgoing edges for each node ID, and a running ID used to allocate new node IDs.

The `root` node is associated with ID zero. Allocation involves assigning a new ID and returning it for use to the user. Inserting an edge $a \to b$ updates the reference count of b and outgoing edge set of a. Deleting an edge $a \to b$ reduces the reference count of b; if it drops to zero, b is placed on the free list and all of its outgoing edges are similarly deleted in a recursive manner.

What is important in this context is not that the GCDS perfectly match the implementation of a real reference counting system (indeed, this would require specializing to a specific language implementation and runtime, which we are trying to abstract away from), but rather ensuring that the performance characteristics we defined earlier (delay and pause time) are not harmed in the translation from real GC approach to GCDS implementation.

Notably, this GCDS does not attempt to store reference counts within the allocated region itself; in fact, it has no notion of "allocated memory" at all because it operates on a graph abstraction of the heap. Intuitively, one can think of the GCDS as operating in a separate address space from the program, seeing only pointer operations as sequences of GCDS operations on handles that are returned by `GCAllocate`. Hence, it must store its own metadata and shadow copy of the graph, e.g., using the 'ref_counts' and 'outgoing' variables in Figure 3. However, it is important to note

Algorithm 1: `GCInit()`

```
ref_counts ← ∅;
outgoing ← ∅;
root ← id ← 0;
```

Algorithm 2: `GCStep()`

```
no-op;
```

Algorithm 3: `GCAllocate()`

```
id ← id + 1;
return id;
```

Algorithm 4: `GCInsert(a → b)`

```
ref_counts[b] ← ref_counts[b] + 1;
outgoing[a] ← outgoing[a] ∪ {b};
```

Algorithm 5: `GCDelete(a → b)`

```
outgoing[a] ← outgoing[a] − {b};
ref_counts[b] ← ref_counts[b] − 1;
if ref_counts[b] = 0 then
    worklist ← {b};
    while worklist ≠ ∅ do
        node ← worklist.pop();
        GCFreeList ← GCFreeList ∪ {node};
        for c ∈ outgoing[node].no_multi() do
            ref_counts[c] ← ref_counts[c] − outgoing[node].count(c);
            if ref_counts[c] = 0 then worklist ← worklist ∪ {c};
```

Fig. 3. Eager reference counting as an implementation of the GCDS interface. ID 0 represents the root node (stack/local/global variables) while all other memory regions get a unique ID upon allocation. `ref_counts` is a map from IDs to counts that could be implemented using a hashmap.

that searching in these variables can be done in $O(1)$ expected time using a hashmap to store the refcounts and outgoing edges, or in $O(1)$ worst-case time using a direct addressing table at the expense of added memory overhead. In general, our impossibility results in this paper apply *regardless of the space usage of the GCDS itself*, hence it is okay if when translating a GC algorithm to implement the GCDS interface more space (even asymptotically more space) is used, as long as the delay and pause times are not affected; all of our results will still apply.

Worst-Case Delay: This reference counting GCDS has no well-defined worst-case delay, because if a sequence of GCDS operations forms a cycle, e.g., `root` $\rightarrow a \rightarrow b \rightarrow c \rightarrow a$, and then deletes `root` $\rightarrow a$, the node a will *never* be added to the free list, no matter how many additional GC operations are performed. We informally write $d(n) = \infty$ for this situation.

Worst-Case Pause Times: In some cases, a single edge removal could make every node in the heap unreachable at once resulting in `GCDelete` having to iterate over every edge in the heap, making $t(n) = O(n^2)$. However, the worst-case *all-reachable* pause time (Definition 3.4) is *significantly* better: if all nodes are still reachable from `root`, then the reference count could not have dropped to zero, and hence the if condition in `GCDelete` is never taken. All the remaining operations take $O(1)$ time, hence $t_{\mathrm{AR}}(n) = O(1)$. By Lemma 3.7 this also implies $t_{\mathrm{A,AR}}(n) = O(1)$ and $t_{\mathrm{S,A,AR}}(n) = O(1)$.

3.3.2 Mark-and-Sweep as a GCDS. Figure 4 shows how the mark-and-sweep technique can be used to implement the GCDS interface. Like the reference counting GCDS, we keep track of a shadow copy of the heap graph. But instead of checking reference counts to determine when something

Algorithm 6: `GCInit()`

outgoing $\leftarrow \emptyset$;
`root` $\leftarrow$ id $\leftarrow 0$;
all $\leftarrow \emptyset$

Algorithm 7: `GCInsert`$(a \to b)$

outgoing$[a] \leftarrow$ outgoing$[a] \cup \{b\}$;

Algorithm 8: `GCAllocate()`

id $\leftarrow$ id $+ 1$;
all $\leftarrow$ all $\cup$ {id};
return *id*;

Algorithm 9: `GCDelete`$(a \to b)$

outgoing$[a] \leftarrow$ outgoing$[a] - \{b\}$;
`GCStep()`;

Algorithm 10: `GCStep()`

worklist $\leftarrow$ {`root`};
marked $\leftarrow \emptyset$;
while worklist $\neq \emptyset$ **do**
    node $\leftarrow$ worklist.pop();
    if node $\in$ marked **then Continue**;
    marked $\leftarrow$ marked $\cup$ {node};
    worklist $\leftarrow$ worklist $\cup$ outgoing[node].no_multi();
`GCFreeList` $\leftarrow$ `GCFreeList` $\cup$ (all $-$ marked);
all $\leftarrow$ all $-$ (all $-$ marked);

Fig. 4. Nonincremental mark-and-sweep as an implementation of the GCDS interface. ID 0 represents the root node (stack/local/global variables) while all other memory regions get a unique ID upon allocation. `outgoing` maps each IDs to a multiset of IDs, while `worklist` is a non-multi set.

becomes unreachable, we perform a search through the graph starting at `root` to determine any newly unreachable nodes.

Worst-Case Delay: This GCDS has worst-case delay $d(n) = 1$, because it guarantees that all unreachable regions are added to the free list *immediately* once they become unreachable.

Worst-Case Pause Times: This GCDS has worst-case pause time $t(n) = O(n^2)$ because the marking phase in `GCDelete` sometimes has to visit every node in the graph, and for each of those nodes must add all of its outgoing nodes to the worklist. This analysis is unaffected by the connectedness or cyclicity of the heap graph, hence we have $t_{\mathrm{AR}}(n) = O(n^2)$ and $t_{\mathrm{A,AR}}(n) = O(n^2)$. But if the heap graph is sparse, then the number of edges is limited and we get $t_{\mathrm{S,A,AR}}(n) = O(n)$.

3.4 Cell Probe Model, Random Access Machines, and Reductions

The existing lower bounds we use [Larsen and Yu 2023; Pătraşcu and Demaine 2004] are in the cell probe model of Yao [1978]. Data structures in the cell probe model are split into a single *persistent store* and a set of *procedures*. The persistent store is a large table of binary words, analogous to the random access memories used in modern computers. The procedures are programs that can read and write words from the persistent store.

Lower bounds in the cell probe model bound only the number of reads from and writes to the persistent store. No assumption at all is made about the machine model that the procedures run on, except that accessing a memory cell takes $\Omega(1)$ time and that writes made to the store are a deterministic function of reads from the store. In fact, lower bounds proved in this way apply even to unrealistic machine models, e.g., Turing machines with an oracle for the halting problem.

One nuance in the cell probe model is the need for a word size for the table. Usually, the data structure is given an upper bound $n_{\max}$ on the number of objects the data structure might be asked to represent, e.g., nodes and edges in a graph or items in a set. On a realistic modern machine, $n_{\max}$ would be approximately 2^{64}. The persistent store is allowed to have a word size logarithmic in this upper bound, i.e., $w = O(\log n_{\max})$.

4 Main Lower Bound

Our goal in this section is to prove that every GCDS has $t(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$. In particular, this means any GCDS guaranteeing $O(1)$ delay must have superlogarithmic pause times. In fact, we will prove the stronger result that $t_{\text{AR}}(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$, i.e., the superlogarithmic pause can be triggered by a sequence of GCDS operations where nothing ever becomes unreachable. This is, at first glance, counterintuitive because if it knew ahead of time that nothing becomes unreachable, the GCDS would not need to do any work. However, the GCDS is not told this ahead of time, and we still require that it guarantee delay $d(n)$ if something *were* to become unreachable. Essentially, the GCDS must still prove to itself that nothing has become unreachable.

4.1 Dynamic Graph Connectivity

Our main lower bound follows via reduction from dynamic graph connectivity, defined below.

Definition 4.1. A *dynamic connectivity data structure* (DCDS) stores a graph (V, E) on a fixed number $n_{\max} = n$ of vertices. It supports the following operations:

(1) `DCInsert(`$a \longrightarrow b$`)` adds edge $a \longrightarrow b$.
(2) `DCDelete(`$a \longrightarrow b$`)` removes edge $a \longrightarrow b$.
(3) `DCConnected(`$a \longrightarrow^{+} b$`)` returns *connected* if and only if there is a path from a to b.

Larsen and Yu [2023] prove that this general form of dynamic connectivity requires $\tilde{\Omega}(\log^{3/2} n)$ time in the cell probe model (Section 3.4) [Yao 1978].

Theorem 4.2. *(Larsen and Yu [2023]) Every DCDS has worst-case $\tilde{\Omega}(\log^{3/2} n)$ per-operation time.*

4.2 Clocked Machines and Checkpoint-Restore

Our reduction involves two special operations: *checkpoint-restore* and *timeouts.*

A ‘restore’ operation causes the state of the data structure’s persistent store to be restored to that of the last ‘checkpoint’ operation. Finitely many checkpoint-restore operations can be implemented with constant overhead on realistic machines by logging memory writes after each checkpoint and undoing them when restoring.

Timeouts stop the operation of a subroutine after a fixed number of instructions are executed. They can be implemented with $O(1)$ overhead by instrumenting the subroutine instructions to continuously increment a clock counter and exit the subroutine if the counter passes the timeout.

4.3 Reduction from Larsen and Yu [2023], DCDS

We now show that an efficient GCDS could be used to construct an efficient DCDS. The existence of such a reduction means that the known lower bounds on the DCDS problem apply to GCDS as well. The reduction is described by Algorithms 11–14.

High-Level Operation of the Reduction. At a high level, we implement the DCDS operations `DCInsert` and `DCDelete` by mirroring their edge manipulations in the GCDS. The GCDS also sees an additional node, X, that has an incoming edge from `root` and an outgoing edge to every other node in the graph. To check for the existence of a path between two nodes, we connect `root` to the

Algorithm 11: DCInsert($v_i \to v_j$)

```
v_i, v_j ← DCCreate(i, j);
GCInsert(v_i → v_j);
```

Algorithm 12: DCDelete($v_i \to v_j$)

```
v_i, v_j ← DCCreate(i, j);
GCDelete(v_i → v_j);
```

Algorithm 13: DCCreate($i_1, i_2, \ldots$)

```
if X = ⊥ then X ← GCAllocate() ;
for i_k ∈ i_1, i_2, ... do
    if i_k ∉ N then
        N(i_k) ← GCAllocate();
        GCInsert(X → N(i_k));
        GCDelete(root → N(i_k));
return N(i_1), N(i_2), ...;
```

Algorithm 14: DCConnected($v_i \to^+ v_j$)

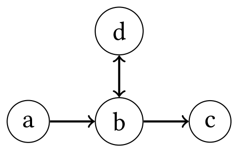
```
CHECKPOINT;
v_i, v_j ← DCCreate(i, j);
GCInsert(root → v_i, v_j → X);
/* The GCDS guarantees that operations take ≤ t_AR(n) time when nothing new
   becomes unreachable, so if it runs longer than that we know something
   must have become unreachable.                                          */
if GCDelete(root → X) runs for more than t_AR(n) instructions or GCFreeList is not empty
 then
    RESTORE;
    return DISCONNECTED;
/* The GCDS guarantees that any unreachable region is collected within d(n)
   operations, so this loop simulates d(n) program operations to guarantee
   that unreachable regions will be detected if any exist.                */
foreach i ∈ 1, 2, ..., d(n) do
    if GCStep() runs for more than t_AR(n) instructions or GCFreeList is not empty then
        RESTORE;
        return DISCONNECTED;
RESTORE;
return CONNECTED;
```

Fig. 5. Reduction from DCDS to GCDS. The DCCreate helper method is needed to lazily create nodes in the GCDS because the DCDS comes preallocated with n nodes while the GCDS begins with none.

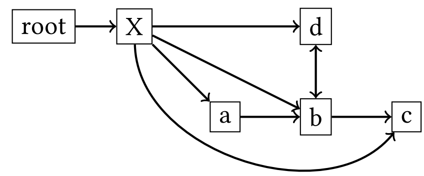
source, connect the destination to X, and then disconnect X from the root. There was a path from source to destination if and only if all nodes remain reachable.

All-Reachable Time Assumptions. Recall our goal is to show the stronger fact that $t_{\text{AR}}(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$, so our reduction we may assume *only* that the all-reachable worst-case pause times are guaranteed to be low. The GCDS is allowed to take an arbitrarily long time—or even refuse to terminate—if there are unreachable nodes. How should we use the GCDS operations, then, if they might cause things to become disconnected? An elegant solution is to timeout the execution of the GCDS (Section 4.2): if it takes more than $t_{\text{AR}}(n)$ steps, it *must* indicate there are unreachable regions. Otherwise, we use its output to determine whether there are any unreachable nodes.

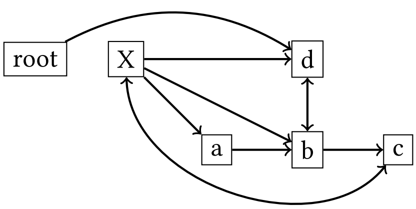
Destructive Operations. The DCConnected reduction requires calling GCInsert and GCDelete, which modify the heap, even though DCConnected is a read-only operation. To address this, we



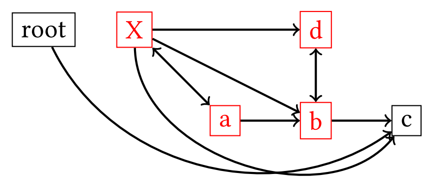
(a) A directed graph for which the DCDS is asked to maintain connectedness information.



(b) The corresponding heap graph constructed in the underlying GCDS used in the reduction.



(c) Checking $d \longrightarrow^+ c$. No nodes become unreachable, so they are connected.



(d) Checking $c \longrightarrow^+ a$. The node a becomes unreachable, so they are disconnected.

Fig. 6. Illustration of the reduction from DCDS to GCDS (Algorithms 11–14).

checkpoint and restore the data structure (Section 4.2) so the persistent store does not see any writes performed during the call to `DCConnected`.

Example 4.3. Figure 6 illustrates our reduction constructing a DCDS given an GCDS. As sketched, the GCDS heap will look like a copy of the DCDS graph, except with an auxiliary node X that has edges from the root and to every other node (compare Figure 6a and Figure 6b). DCDS insertion and deletion operations are passed directly to their corresponding GCDS operation. To check for a path $d \longrightarrow^+ c$, we add edges `root` $\longrightarrow d$ and $c \longrightarrow X$, then `root` $\longrightarrow X$ (Figure 6c). If there is a path $d \longrightarrow^+ c$, then c will remain reachable from `root` and hence so will X and hence so will every other node (Figure 6c). Otherwise, if there is no such path, as in Figure 6d where we have checked for a path $c \longrightarrow^+ a$, at least the node X will become unreachable and so `GCDelete` will either report unreachable nodes or time out. In either case, we know whether the path exists or not. □

We can now prove the reduction correct.

THEOREM 4.4. *Suppose a GCDS has all-reachable worst-case pause times $t_{\text{AR}}(n)$ and worst-case delay $d(n)$ (Definitions 3.4, 3.2). Then there exists a DCDS taking time $O(t_{\text{AR}}(n)d(n))$ per operation.*

PROOF. Figure 5 constructs such a DCDS assuming access to such a GCDS. `DCInsert` and `DCDelete` call the corresponding GC operation, ensuring the GCDS sees a mirror of the graph.

`DCConnected`($v_i \longrightarrow^+ v_j$) is slightly more complicated. We insert the edge `root` $\longrightarrow v_i$ and remove the edge `root` $\longrightarrow X$. Hence, the root is directly connected *only* to v_i, and v_j will be reachable from `root` if and only if there is a path $v_i \longrightarrow^+ v_j$. But because $v_j \longrightarrow X$ and X has an edge to every other node, the entire heap remains reachable if and only if there is a path $v_i \longrightarrow^+ v_j$. Hence, after timing out `GCDelete` in order to use it as a binary decider whether any node has become disconnected, the result tells us whether there is a path $v_i \longrightarrow^+ v_j$. □

Note that the theorem holds even if we do not know the precise expression for $t_{\mathrm{AR}}(n)$ and $d(n)$; as long as we know an asymptotic upper bound, it is *possible* (in theory) to carry out the construction in Figure 5 and hence the lower bound holds. We now note the following important corollaries:

Corollary 4.5. *Any correct GCDS must have* $t_{\mathrm{AR}}(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$.

Proof. By the prior theorem and that of Larsen and Yu [2023]. □

Corollary 4.6. *Any correct GCDS must have* $t(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$.

Proof. By the prior corollary and Lemma 3.7. □

4.4 Hints to the Data Structure

Our reduction has the following property: if the DCDS graph is acyclic, then all cycles in the heap seen by the GCDS go through the $X \leftrightarrow v_j$ edge. Because the lower bound from Larsen and Yu [2023] holds even when restricted to acyclic graphs, no GCDS can beat the $\tilde{\Omega}(\log^{3/2} n)$ lower bound *even if* the data structure is told that all cycles are cut by a single edge, and given that edge. This is an example where weak–strong pointer annotations do not help: the edge is necessary to reach many nodes, hence it cannot be weak, but also inherently introduces cycles, hence it cannot be strong.

4.5 Conditional Lower Bounds

The graph algorithms community has proposed a problem, the *online matrix-vector multiplication problem* (OMV) [Henzinger et al. 2015], that they conjecture to be hard. Similar to the exponential time conjecture about SAT, the OMV conjecture implies other problems should have certain lower bounds as well. In particular, the OMV conjecture would imply that the DCDS problem is lower bounded by $\Omega(n)$ and hence it would imply any GCDS has $t_{\mathrm{AR}}(n)d(n) = \Omega(n)$ and therefore also that $t(n)d(n) = \Omega(n)$.

5 Sparse, Acyclic Lower Bound

In the last section we proved $t_{\mathrm{AR}}(n)d(n) = \tilde{\Omega}(\log^{3/2} n)$, implying that for any GCDS with $d(n) = O(1)$ there is a sequence of operations where (i) one of the operations takes time at least $\tilde{\Omega}(\log^{3/2} n)$ and (ii) none of the operations makes anything unreachable. However, no clear guarantee was made about the shape of the heap in that worst-case-triggering sequence, e.g., triggering the $\tilde{\Omega}(\log^{3/2} n)$-time pause might require a heap with cycles or many outgoing edges from each region.

To address these concerns we prove in this section that $t_{\mathrm{S,A,AR}}(n)d(n) = \Omega(\log n)$. In other words, for any GCDS with $O(1)$ worst-case collection delay there is a sequence of operations (in fact, a family of sequences of operations, one for each n) where:

(1) One of the operations takes time at least $\Omega(\log n)$;
(2) None of the operations makes anything unreachable;
(3) At all points, the heap is acyclic; and
(4) At all points, the heap is sparse.

(Section 7 will prove that the reduction in bound from $\log^{3/2} n$ to $\log n$ is unavoidable.)

5.1 Layered Permutation Graph Connectivity

We are not aware of any way to prove the desired result using the result of Larsen and Yu [2023]. Instead, this section presents a reduction from the *layered permutation connectivity* problem, defined on the next page.

Definition 5.1. A *layered permutation connectivity data structure* (LPCDS) is identical to a DCDS (Definition 4.1) except each vertex is assigned a layer number and the caller promises:

(1) Edges will only be inserted from one layer to the subsequent one,
(2) Connectedness queries always start from the first layer, and
(3) Every vertex has indegree and outdegree at most one.

To distinguish LPCDS operations from DCDS, we call them `LPCInsert`, `LPCDelete`, `LPCConnected`.

Pătraşcu and Demaine [2004] prove that any such data structure requires at least $\Omega(\log n)$ time in the cell probe model (Section 3.4) [Yao 1978], even when amortized over arbitrarily long sequences of operations. The presentation here differs slightly from that of Pătraşcu and Demaine [2004] because they consider undirected graphs and do not provide the algorithm with the layer information directly; Appendix C explains how lower bounds from their work apply to this problem as well.

Theorem 5.2. *([Pătraşcu and Demaine 2004]) Every LPCDS has worst-case $\Omega(\log n)$ per-operation time.*

5.2 Our Reduction

We will assume access to a GCDS that is efficient on sequences of operations where nothing becomes unreachable and the heap remains sparse and acyclic. We must use this GCDS to build an efficient algorithm for the LPCDS problem.

The reduction we will use is very similar to the general one described in Section 4.3. However, that reduction relied fundamentally on the insertion of a cycle, namely, $X \leftrightarrow v_j$, so it seems unclear how to make use of the assumption about a time bound that applies only when the heap is acyclic. The key is to use a slightly different reduction, shown in Algorithms 15–18. The below example illustrates this reduction.

Example 5.3. Consider the layered permutation graph in Figure 8a. Our goal is to use an efficient GCDS to quickly answer connectedness queries in such graphs. Our reduction makes a copy of the layered permutation graph in the GCDS, and connects `root` to every node with no incoming edges so the graph remains connected (Figure 8b). To check for a path $a \to^+ b$, we add an edge from $b \to a$ and then disconnect a from `root`. If they *are* connected, i.e., they are on the same path (Figure 8c), then a cycle will be inserted and both will become unreachable. If they *are not* connected, i.e., they belong to different paths, then b will remain connected and hence so will a and everything else along its path without the introduction of any cycles (Figure 8d). In either case, the presence of cycles or unreachable nodes tells us whether a path exists. Because we are assuming the GCDS is fast when no cycles are present and nothing becomes unreachable, we can use the same timeout idea to quickly check for these conditions. □

We can now prove the main theorem of this section.

Theorem 5.4. *Suppose some GCDS has sparse, acyclic, all-reachable worst-case pause times $t_{\text{S,A,AR}}(n)$ and worst-case delay $d(n)$. Then there exists an LPCDS taking time $O(t_{\text{S,A,AR}}(n)d(n))$ per operation.*

Proof. The algorithms for the reduction are given in Figure 7. We mirror the graph into the GCDS heap, but add an edge from `root` to every node with indegree zero. This ensures every node is reachable via a unique path from `root`.

To check if $v_i \to^+ v_j$, i.e., whether v_j is in the path starting at v_i, we add $v_j \to v_i$ then disconnect v_i from `root`. If they *are* connected, we introduced a cycle and severed the unique path to v_i and v_j. Otherwise, no cycle is inserted and v_i remains accessible via the path from v_j. In either case, the algorithm correctly reports the (non)existence of such a path to the caller. □

Algorithm 15: LPCInsert($v_i \rightarrow v_j$)

$v_i, v_j \leftarrow$ LPCCreate(i, j);
GCInsert($v_i \rightarrow v_j$);
GCDelete(root $\rightarrow v_j$);

Algorithm 16: LPCDelete($v_i \rightarrow v_j$)

$v_i, v_j \leftarrow$ LPCCreate(i, j);
GCInsert(root $\rightarrow v_j$);
GCDelete($v_i \rightarrow v_j$);

Algorithm 17: LPCCreate($i_1, i_2, \ldots$)

for $i_k \in i_1, i_2, \ldots$ **do**
  if $i_k \notin N$ **then**
    $N(i_k) \leftarrow$ GCAllocate();
return $N(i_1), N(i_2), \ldots$;

Algorithm 18: LPCConnected($v_i \rightarrow^+ v_j$)

Checkpoint;
$v_i, v_j \leftarrow$ LPCCreate(i, j);
GCInsert($v_j \rightarrow v_i$);
if GCDelete(root $\rightarrow v_i$) *runs for more than* $t_{\text{S,A,AR}}(n)$ *instructions or* GCFreeList *is not empty* **then**
  Restore;
  return *Connected*;
foreach $i \in 1, 2, \ldots, d(n)$ **do**
  if GCStep() *runs for more than* $t_{\text{S,A,AR}}(n)$ *instructions or* GCFreeList *is not empty* **then**
    Restore;
    return *Connected*;
Restore;
return *Disconnected*;

Fig. 7. Reduction from LPCDS to GCDS.

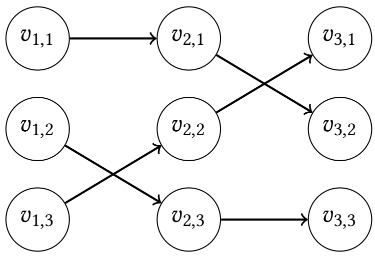
Hence, we immediately get the following two corollaries:

Corollary 5.5. *Any correct GCDS must have* $t_{\text{S,A,AR}}(n)d(n) = \Omega(\log n)$.

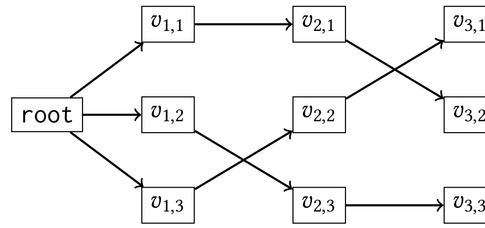
Proof. By the prior theorem and that of Larsen and Yu [2023]. □

5.3 Connection to Cyclic Reference Counting

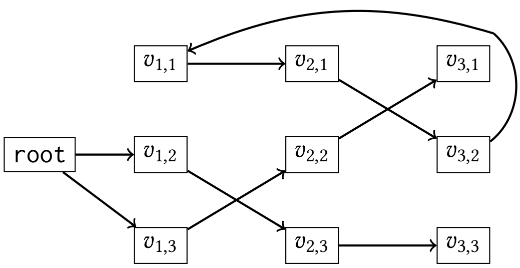
The results in this section connect closely to the field of *cyclic reference counting*, which is focused on extending reference counting to support heaps with cycles. One of the first algorithms, from Brownbridge [1985], claimed to support immediate collection of all unreachable regions, even in the presence of cycles, while ensuring $O(1)$ pause times for acyclic heaps, i.e., violating Corollary 5.5. But it was discovered to be incorrect by Salkild [1985]. If the lower bounds in this section were known at the time, sanity checking that something is wrong with the claims would have been significantly easier. More recently, the English translation of Pepels et al. [1988] also claims to beat our lower bound, i.e., guarantee immediate collection for all heaps and also guarantee $O(1)$ pause times when the heap is acyclic. We believe this claim is simply a mistranslation; nonetheless, we provide in Appendix B a counterexample to the claim.



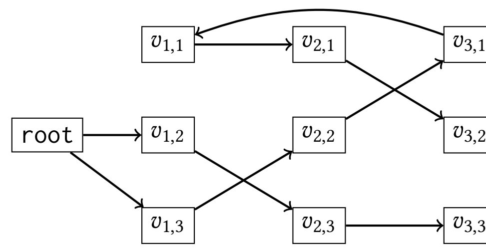
(a) A layered permutation graph as in Pătraşcu and Demaine [2004].



(b) The corresponding heap graph we use for the lower bound reduction.



(c) Checking $v_{1,1} \to^+ v_{3,2}$. A cycle is introduced and $v_{1,1}$ becomes unreachable, so they are connected.



(d) Checking $v_{1,1} \to^+ v_{3,1}$. The graph remains acyclic and nothing becomes unreachable, so they are disconnected.

Fig. 8. Illustration of the reduction from the problem of Pătraşcu and Demaine [2004] to GCDS.

6 Application-Level Implications of Lower Bounds

We now briefly review the application-level consequences of the lower bounds in the last two sections. Consider a programming language with a garbage collector implementing the relatively standard GCDS interface, i.e., in an abstract sense, it sees the program as a sequence of pointer insertions and removals.

6.1 Main Lower Bound, Existence of Real-Time Violations

Suppose the language enforces a memory limit M. Suppose the underlying collector enforces an $O(1)$ pause time limit, i.e., $t(n) = O(1)$. Theorem 4.4 and Corollary 4.5 imply there exists at least one program of the following form:

(1) First fill the memory up to the limit with equal-sized allocations and perform some pointer manipulations while keeping every region reachable;
(2) Then, overwrite a pointer to make some region unreachable (collectable);
(3) Then, request a new allocation;

such that the region made collectable in the second step takes $\tilde{\Omega}(\log^{3/2} M)$ program steps to collect. Hence, in the third step, either the program must introduce a $\tilde{\Omega}(\log^{3/2} M)$-time pause to complete collection, or it must incorrectly report out-of-memory. Either case could be catastrophic in real-time contexts.

6.2 Acyclic Lower Bound

Suppose again that the language attempts to guarantee for the user that finalizers are called reliably and promptly, i.e., $d(n) = O(1)$. The second lower bound also implies that there must exist a program, even a program where nothing ever becomes unreachable *and the heap stays sparse and acyclic*, where this language introduces an $\Omega(\log n)$-length pause time after some program operation. Equivalently, reference counting can not be extended to handle cycles without slowing down execution on some program that only makes acyclic heap structures.

7 Upper Bound

Section 5 showed that any GCDS guaranteeing constant collection delay for all programs ($d(n) = O(1)$) must introduce significant slowdown on some program making only a sparse, acyclic heap, even if that program never makes any region unreachable ($t_{\text{S,A,AR}}(n) = \Omega(\log n)$).

This section shows that this bound is tight: there is a GCDS that guarantees constant collection delay for all programs ($d(n) = 1$), and guarantees that no operation takes longer than $O(\log n)$ time when the heap is acyclic and nothing becomes unreachable ($t_{\text{A,AR}}(n) = O(\log n)$). In fact, we will eventually show the stronger fact that, when the heap is acyclic, our GCDS has exactly an $O(\log n)$-factor slowdown compared to reference counting.

The key difference with reference counting, however, is that our algorithm still guarantees constant collection delay even in the presence of cycles. Combined with the lower bound in the prior section, this algorithm has optimal acyclic, all-reachable worst-case pause times among all algorithms that guarantee $d(n) = O(1)$. However, because it involves both frequent balanced tree operations and increased (though still constant sized) metadata per allocation region, we have found that it is not competitive in non-pathological cases with traditional garbage collection algorithms. Hence, we do not recommend it as a general-purpose GC; its primary purpose is to answer the theoretical questions described above.

The algorithm itself is adapted from similar algorithms for dynamic connectivity in undirected graphs [Holm et al. 2001; Kapron et al. 2013]. Like those, the key idea is to maintain a spanning tree of the reachable heap in an Euler tour data structure (Section 7.1). The main complication is updating the spanning tree when an edge in the spanning tree is removed from the heap. Prior work has devised many clever techniques for quickly updating the spanning tree when the graph is undirected, however, these techniques are not immediately applicable to the directed case needed for proper GC. Instead, we identify a strategy for updating spanning trees of directed graphs that is sufficient to guarantee the $O(\log n)$ time when the graph is acyclic.

7.1 Euler Tour Data Structure

Our algorithm relies on the Euler tour data structure, which is a well-known data structure for efficiently storing and manipulating directed forests. A *directed tree* is a graph (V, E) that has exactly $|V| - 1$ edges and a distinguished *root* vertex that can reach every other node. A *directed forest* is a disjoint union of directed trees. The Euler Tour data structure stores a directed forest while allowing efficient edge insertions, edge deletions, and connectedness queries.

Definition 7.1. The Euler Tour data structure (ETDS) stores directed forests and supports:

- `ETSingleton`() returns a new singleton tree in the forest.
- `ETCut`($a \to b$) splits a tree into two by removing the edge $a \to b$.
- `ETLink`($a \to b$) links a tree containing a and one rooted at b by inserting the edge $a \to b$.
- `ETPath`($a \to^+ b$) is true if and only if there is a directed path from a to b in the forest.
- `ETParent`(a) returns the immediate parent of a in the tree.
- `ETNext`(a) returns the node after a in the preorder traversal of the tree.

There exist deterministic algorithms for the Euler tour data structure where `ETSingleton`, `ETCut`, `ETLink`, and `ETPath` all take time $O(\log n)$ while `ETNext` and `ETParent` take time $O(1)$. The insight

Algorithm 19: GCInit()

```
root ← ETSingleton();
```

Algorithm 20: GCInsert(a → b)

```
E(a → b) ← E(a → b) + 1;
```

Algorithm 21: GCAllocate()

```
n ← ETSingleton();
ETLink(root → n);
E(root → n) ← 1;
return n;
```

Algorithm 22: GCStep()

```
no-op;
```

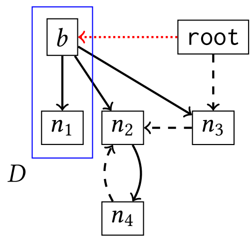
Algorithm 23: GCDelete(a → b)

```
E(a → b) ← E(a → b) − 1;
if E(a → b) > 0 or ETParent(b) ≠ a then return ;
ETCut(a → b);
while ⊤ do
    D ← ∅ ;                    /* Nodes earlier in the preorder traversal */
    c ← ⊥ ;                /* Continue iterations, or reached fixedpoint? */
    /* Iterate over the tree rooted at b; keep two pointers so we can
       continue the iteration even if a subtree is cut out.            */
    p, n ← ⊥, b;
    while n is not none do
        /* Look for a new parent either in the main spanning tree or later
           in the preorder traversal.                                  */
        Find m such that E(m → n) > 0, m ∉ D, and ¬ETPath(n →+ m);
        if there is an m satisfying those conditions then
            ETCut(n) ;  /* Cut subtree away from the separated spanning tree */
            ETLink(m → n) ;                 /* Reconnect it to the new parent */
            if n = b then return;
            if ETPath(root →+ n) then c = ⊤;
            p, n ← p, ETNext(p) ;                        /* Skip that subtree */
        else
            D ← D ∪ {n} ;                         /* Do not link later nodes here */
            p, n ← n, ETNext(n) ;          /* Continue the preorder traversal */
    if c = ⊥ then
        GCFreeList ← GCFreeList ∪ D;
        return
```

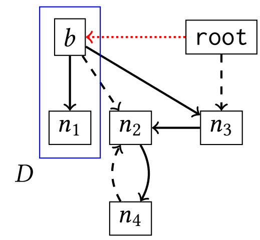
is to represent each tree by its *Euler tour*. This Euler tour is itself stored in a balanced binary tree. See Tarjan and Vishkin [1984] for a detailed discussion of the algorithms.

7.2 Algorithm and Pseudocode

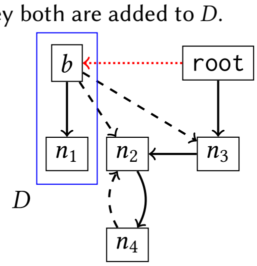
For ease of exposition the majority of the section describes a simplified algorithm that only guarantees short worst-case pause times when the heap is both sparse and acyclic. Then in Section 7.4 we describe minor optimizations that permit the same guarantees when the heap is dense. The proposed GCDS is described by Algorithm 21–23. It stores two pieces of information:



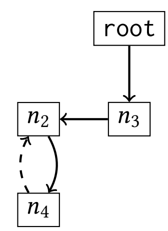
(a) Here, the algorithm already visited b and n_1. Neither had an incoming edge from a node not in D or from the main spanning tree (which is now just `root`), so they both are added to D.



(b) The only edge to n_2 that is from a node neither in D nor a tree-descendant of n_2 is $n_3 \longrightarrow n_2$. Hence, we update the spanning tree by cutting n_2 from its parent and linking it to n_3.



(c) A similar analysis finds that n_3 can be rooted at `root` $\longrightarrow n_3$, so the spanning tree is updated. This links n_3 and its entire subtree, including n_2 and n_4, to the main spanning tree.



(d) A final iteration of the loop finds that nothing else can be connected to the main spanning tree, hence $D = \{b, n_1\}$ can be deleted from the graph and freed in the program.

Fig. 9. Visualizing the operation of `GCDelete(root` $\longrightarrow b$`)`. The red dotted edge is the deleted edge. The solid edges represent edges in the spanning trees. The dashed edges represent edges in the heap graph that are not currently part of the spanning tree. The visualization starts after two nodes in the preorder have already been visited. We then visit n_2, relinking it to n_3. After visiting n_3 and relinking it to `root`, the maximal spanning tree is restored and the remaining nodes can be freed. The blue box represents the set D from Algorithm 23.

(1) An Euler Tour Tree representation of a spanning forest for the heap graph, and
(2) A map from edges to multiplicity that can be queried for all edges ending in a given node.

To allocate a new node, we create the node in the ETT, then link that ETT node to the one for `root`. Inserting an edge does not need to update the spanning tree because we know the node was already reachable from `root` and hence already part of the spanning tree.

Deleting an edge $a \longrightarrow b$ is more difficult. If a is *not* the parent of b, i.e., $a \longrightarrow b$ is not in the spanning tree, we simply remove it from the edges map and leave the tree unchanged. Otherwise, if $a \longrightarrow b$ was an edge in the spanning tree, we cut b from the spanning tree and begin a preorder iteration of the newly separated tree rooted at b. For each node n, we iterate over its incoming edges $m \longrightarrow n$ looking for one where m is not a descendant of n and either (i) m is part of the main spanning tree, or (ii) m is in the separated tree but not yet visited. If an incoming edge $m \longrightarrow n$ satisfying those two conditions is found, we cut the subtree rooted at n out of the tree and relink it at m. Intuitively, this process pushes subtrees either back into the main spanning tree or further down the preorder traversal of the newly separated tree. We repeat this sweeping operation until the entire sweep reveals no nodes with incoming edges from the main spanning tree, at which point we know the entire separated tree is unreachable. (Note `ETNext`$(p) = \bot$ only when $n = b$, hence line 15 will never be reached with `ETNext`$(p) = \bot$ — it would instead return on line 13.)

Example 7.2. The execution of `GCDelete` is visualized in Figure 9. We start with a heap having six nodes, where the spanning tree is made up of edges `root` $\to b$, $b \to n_1$, $b \to n_2$, $b \to n_3$, and $n_2 \to n_4$. Then, the operation `GCDelete(root` $\to b$`)` is performed.

First, the algorithm visits b, finding that, after removing the edge `root` $\to b$, it has no other incoming edges at all. Hence, it is added to the set D and we continue the preorder sweep to its child n_1. The only incoming edge to n_1 is $b \to n_1$, but $b \in D$ so we also add n_1 to D and continue the sweep. The resulting state is illustrated in Figure 9a.

Next in the preorder is node n_2. The incoming edge $b \to n_2$ is not usable because $b \in D$. Meanwhile, the edge $n_4 \to n_2$ is not usable because n_4 is a descendant of n_2 in the spanning tree — adding that edge would cause a cycle. Finally, we find that the edge $n_3 \to n_2$ works, so we unlink n_2 from b and reconnect it to the spanning tree via n_3. The resulting state is illustrated in Figure 9b.

Finally we visit n_3. The edge from b is unusable because $b \in D$, but the edge from `root` can be used. This reconnects n_3 to the main spanning tree, which now contains n_3, n_2, and n_4. The resulting state is illustrated in Figure 9c.

Because we have reconnected something to the main spanning tree during this sweep, we must resweep across the remaining two nodes in the spanning tree rooted at b (b and n_1). Neither can be reconnected to the main spanning tree, i.e., the one rooted at `root`, so the algorithm completes by removing those nodes and adding them to `GCFreeList`. □

With the operation of the algorithm now illustrated, we can prove correctness and termination.

THEOREM 7.3. *Algorithms 19–23 terminate and correctly implement a GCDS with* $d(n) = 1$.

PROOF. The algorithm maintains a maximal spanning tree from `root`. This property follows immediately for `GCInit()`, `GCInsert()`, and `GCAllocate()`. To see correctness of `GCDelete`, first note we only ever call `ETLink` on edges that actually exist in the graph, so the spanning tree is at least a subset of the maximal spanning tree. Second, note that a node is only added to D if it has no incoming edges from the main spanning tree, and the algorithm only halts when D contains all the nodes in the newly separated tree. Hence, there can be no edges from the main spanning tree to any node in the newly separated tree, i.e., the main spanning tree is indeed maximal.

For termination, it suffices to note that on every iteration of the outer loop except the last at least one node is moved to the main spanning tree. To see that $d(n) = 1$, it suffices to note that `GCDelete` eagerly adds any newly unreachable nodes to the free list. □

7.3 Sparse, Acyclic, All-Reachable Worst-Case Pause Time

We now prove that the GCDS has $t_{\text{S,A,AR}}(n) = O(\log n)$, i.e., worst-case logarithmic pause times when the heap is sparse and acyclic and nothing becomes unreachable. The time bounds for all operations except for `GCDelete` follow immediately from those of the underlying data structures, so we primarily focus on `GCDelete`.

We can prove the stronger fact that, when the heap is sparse and acyclic, `GCDelete` runs in time a logarithmic factor slower than the number of newly unreachable edges. Because reference counting runs in time proportional to the number of newly unreachable edges, this implies our GCDS runs with only a logarithmic factor overhead compared to eager reference counting as long as the heap is sparse and acyclic. We will use the symbol Δ for the number of edges that become unreachable; this is identical to the number of counter decrements that reference counting would have to do.

THEOREM 7.4. *If* b *cannot reach any cycles, every node reachable from* b *has* $O(1)$ *indegree, and* Δ *edges become unreachable after removing* $a \to b$*, then* `GCDelete(`$a \to b$`)` *runs in time* $O(\Delta \log n)$.

PROOF. In the acyclic case, `ETPath(`$n \to^{+} m$`)` is impossible. Hence, a node is only added to D when all of its predecessors are in D. Because D is initially empty, induction ensures D contains only nodes that are now unreachable from `root`. This implies the outer loop iterates exactly twice, as the first iteration identifies and links all still-reachable nodes back to the main spanning tree.

Because we skip over subtrees when a new parent is found, we only visit a node when its parent was added to D. In that case, the parent is unreachable so the edge from the parent to the node is counted by Δ. This ensures we do not visit any node too many times. Finally, thanks to sparsity, only $O(1)$ incoming edges must be analyzed for each visited node. Each such iteration costs $O(\log n)$ to check for the existence of a path. Together, these imply the desired $O(\Delta \log n)$ time complexity. □

We now note the following corollary:

Corollary 7.5. *The GCDS guarantees* $d(n) = 1$ *and* $t_{\text{S,A,AR}}(n) = O(\log n)$.

Proof. The delay claim follows from Theorem 7.3. The time follows from Theorem 7.4 ($t_{\text{S,A,AR}}(n)$ only counts operations where nothing becomes unreachable, i.e., $\Delta = 1$ in Theorem 7.4). □

7.4 Handling Dense Heaps

The above time bound is within an $O(\log n)$ factor of reference counting when the graph is sparse and acyclic. The sparsity requirement can be eliminated by memoizing edge information when searching for a new parent. If the same node is visited twice along a single sweep, any incoming edges that were invalid during the first time it was visited are still invalid the second time. Hence, we can keep a pointer for each node to the last edge we examined, avoiding revisiting edges in later iterations. This guarantees, in the acyclic case, that you only look at truly dead edges, i.e., at most $O(\Delta)$ edges, for an overall time $O(\Delta \log n)$, a log-factor overhead compared to reference counting.

7.5 Memory Overhead

When a node is added to the free list, it will never be referred to again. Hence, `GCDelete` can free any related internal data structure memory, e.g., the corresponding Euler tour tree nodes. The overall scheme then has constant-factor memory overhead: every nonfreed memory region has a corresponding ETT node, and every heap pointer has a corresponding entry in the edge map.

7.6 Exploratory Implementation and Evaluation

This paper is primarily theoretical, but we wondered how the above algorithm would perform in practice. We implemented a variant of it as a GC for Lua. Lua was a convenient choice for implementation as it is a popular language with a relatively simple implementation. It uses a tricolor tracing GC, which is considerably more involved than the naïve eager mark-and-sweep approach but significantly less sophisticated than modern GCs used by Java, Javascript, etc.

We also compared to an existing GC algorithm that can be configured to guarantees $O(1)$ collection delay, namely, we translated Figure 2 ("Synchronous Cycle Collection," SynCC) of [Bacon and Rajan 2001] into C and ran its `CollectCycles` routine after every pointer decrement. Notably, this always-collect configuration is not suggested by [Bacon and Rajan 2001], but it does guarantee immediate collection in the presence of cycles (the same guarantee our ETT-based algorithm makes). Experiments were performed on an Intel(R) Core(TM) i9-13900, with 32 GB of memory running Debian 12 and the `jemalloc` allocator. Other evaluation details are in Appendix E.

7.6.1 Motivating GC Thrashing Example. Recall the motivating example from Section 2.1, where frequent collections were needed in a memory-constrained scenario. The resulting GC thrashing increased the time from approximately 1.1 seconds for a version of the program using manual memory management to almost 70 seconds when the limit was enforced. We reran the same benchmark with both SynCC and our proposed GC. SynCC took approximately 1.1 seconds while ours took 1.4 seconds. SynCC performs better because the heap is simple and does not have any

Table 1. Times are averaged over 5 runs. Overhead is reported as a ratio compared to the default Lua interpreter; 1× means no overhead. "Nonimmediate" is the default Lua collector, while "SynCC" is another collector also guaranteeing $d(n) = 1$ (see description in Section 7.6). Guaranteeing prompt collection usually takes additional time, but our approach was faster than SynCC. Our approach has higher memory overhead due to additional metadata overhead per allocation region to store the ETDS. The reader should be careful to note that the default Lua collector ("Nonimmediate") is relatively naïve compared to the standard Java, JavaScript, etc., collectors, hence in corresponding experiments for those languages we would expect the "Nonimmediate" times and memory usage to be lower and the overhead columns to be higher.

	Nonimmediate		SynCC Overhead		Our Overhead	
Benchmark	Time (s)	Mem (B)	Time	Mem	Time	Mem
`binarytrees-15`	0.3692	13044566.00	3.1642×	1.2950×	2.1207×	2.9869×
`helloworld`	0.0016	97413.00	0.9957×	1.1572×	0.9874×	1.6261×
`merkletrees-15`	0.9468	20163331.00	3.3933×	1.6099×	1.7635×	3.5153×
`nbody-100000`	0.0973	108750.00	1.6883×	1.1982×	0.9889×	1.7376×
`specnorm-1000`	0.8710	147047.00	1.5239×	1.1336×	1.2688×	1.4862×
`list-4096`	0.0026	1537362.00	1562.9055×	1.6088×	1.6452×	3.4310×
`dbllist-4096`	0.0033	2061658.00	2742.9327×	1.5811×	1.9720×	3.1942×

long pointer chains; in Section 7.6.2 we will see scenarios where we beat SynCC by multiple orders of magnitude. This example clearly demonstrates a need for fast, prompt collection and shows that GCs with constant delay can sometimes improve performance over a traditional tracing collector.

7.6.2 General Benchmarks. Finally, Table 1 shows the timing results for multiple Lua benchmarks, most taken from [plb 2023]. The default Lua collector usually has better end-to-end time than both of the immediate collectors. This is unsurprising, because it makes no guarantees about when or whether regions will be collected and finalizers will be called. In fact, the Lua interpreter is partially optimized for implementation size and simplicity so its GC is naïve vs. those in Java, Javascript, etc. Hence, we would expect on such languages there would be an even wider gap between the default collector's performance and the two immediate collectors.

Between the two collectors guaranteeing $d(n) = 1$, our time overhead is consistently better than that of SynCC. This is highlighted by the `list-4096` benchmark, which builds and then traverses a long linked list (the `dbllist-4096` is similar, except uses a doubly linked list). Our technique guarantees an asymptotic $O(\log n)$ factor overhead for such acyclic structures. On the other hand SynCC must scan nearly the entire heap after every operation on a node, introducing an asymptotic $O(n)$ factor overhead on program operations. Notably, our approach has a comparably high memory overhead due to the need to store metadata (splay tree nodes for the ETDS). Although this memory overhead is a constant factor of the number of allocations, it is larger than that needed for reference counting or nonimmediate collection. Hence we do not suggest our algorithm as a good general-purpose collector, but rather an interesting and nonobvious point in the space of asymptotic tradeoffs that could perhaps inspire future applications.

8 Limitations, Open Problems, and Future Work

The exact worst-case GCDS pause time–delay tradeoff is left open by our paper. Appendix F provides a reduction from GCDS to DCDS implying any improved GCDS lower bounds would immediately give stronger DCDS bounds. Hence, because lower bounds for the DCDS problem have resisted significant improvement despite a large amount of research effort, it is unlikely that significantly improved GCDS bounds can be found without novel lower bound techniques.

Our analysis focused on collection delay, motivated by pathological cases where modern GCs perform poorly when additional memory must be found quickly. It would be interesting to explore other notions of garbage collection performance and bounds on the performance of compacting collectors. Many compacting collectors are susceptible to our bounds because they can be converted into inplace collectors [Baker 1992], but we have not formalized the extent of this connection.

We focused on asymptotic time complexity, ignoring memory overhead. Our upper bound requires only a constant factor memory overhead, but the constant is larger than for reference counting. It would be interesting to determine a fine-grained tradeoff lower bound between the constant factor memory overhead required and the time overhead.

9 Related Work

See Knuth [1997] for various garbage collection techniques and their running time, and Henzinger et al. [2015] for a survey of recent bounds for dynamic graph algorithms.

Early Garbage Collection Research. The need for GC was noticed almost immediately after the development of the linked list data structure [Newell and Shaw 1957]. This led to a flurry of work on GC, including reference counting [Collins 1960; Gelernter et al. 1960], mark-and-sweep [McCarthy 1960], hybrids [McBeth 1963], and copying collectors [Minsky 1963].

Tradeoffs between pause times, predictability, and leakage in different GC algorithms have been debated since these early days. Weizenbaum [1962] publicized the failure of reference counting to avoid leaks in the presence of cycles, but his solution had long pause times [McBeth 1963]. Weizenbaum [1964] suggested a hybrid collector, but pathological programs causing poor collector performance remain to this day. We provide a precise, formal proof showing some such tradeoffs are unavoidable.

Concurrent and Real-Time Garbage Collection. Modern GC can run concurrently with the program [Bacon et al. 2003; Baker 1992; Jr. 1978; Lieberman and Hewitt 1983; Minsky 1963]. Many are presented as so-called *real-time* collectors, meaning under certain well-defined scenarios they can guarantee pause times do not exceed a certain constant limit and new allocations can always be serviced [Baker 1992; Jr. 1978]. Per the lower bounds in this paper, all such schemes are susceptible to the sort of pathological examples described in Section 2, where, e.g., programs operating close to the memory limit must either reject allocations unnecessarily or introduce long pauses. For example, Appendix D walks through an example program for which a classic real-time collector would have to either reject allocations unnecessarily or introduce large pause times. In practice, this is avoided by overprovisioning memory, because many real-time collectors can still make guarantees relating the peak logical memory usage to the peak actual memory usage. But overprovisioning resources is expensive, and difficult to do when memory limits depend on other programs sharing resources.

Cyclic Reference Counting. Many have attempted to modify reference counting to handle cycles without requiring a full pass through the heap. A key paper, Salkild [1985], seems to be unavailable today so our discussion about it is based on second-hand reports from other citations here.

Brownbridge [1985] proposed one approach in 1985, but Salkild [1985] soon discovered it might free reachable memory. Apparently Salkild proposed a solution that did not guarantee termination, and eventually Pepels et al. [1988] had found and proved correct a correct and terminating cyclic reference counting scheme but no tight running time analysis was performed. Nowadays, algorithms based on the local mark-scan of [Martínez et al. 1990] are more common in the literature, such as Lins [1992] who removes the immediacy of cyclic reference counting. The similarity between such algorithms and traditional mark-scan is observed by Bacon et al. [2004].

As described in Section 5.3, Sections 7 and 5 answer a longstanding open question in cyclic reference counting: can reference counting be modified to always collect cycles, while guaranteeing reference counting-like overhead when the heap is acyclic? Section 5 says *no*: a logarithmic slowdown is inevitable. Section 7 says *only* a logarithmic slowdown is needed.

Languages Supporting Garbage Collection. Popular implementations of the Java, Javascript, Lua, Python, and Go languages all feature mature concurrent and/or generational mark-and-sweep collectors [Go [n. d.]; Ierusalimschy [n. d.]; Marshall [n. d.]; Oracle [n. d.]; Salgado and Katriel [n. d.]]. The Java language has been a particularly popular platform for GC research [Beronic et al. 2022; Grgic et al. 2018a,b; Mao et al. 2009]. Garbage collectors also exist for the C and C++ languages [Boehm 1993; Rafkind et al. 2009].

Research on New Collection Algorithms. A particularly active area of modern research is in designing collectors that work efficiently in the concurrent or distributed setting [Clebsch and Drossopoulou 2013; Gog et al. 2015; Kang and Jung 2020; Kim et al. 2014] and in using different heuristics than the standard generational approach [Hirzel et al. 2003]. Machine learning might be able to improve GC performance in some common cases [Cen et al. 2020].

Analyzing Garbage Collection. Most authors seem to implicitly understand and take for granted that any collector will have pathological cases with poor worst-case performance [Bacon et al. 2004; Knuth 1997], and focus instead on computing, e.g., the amount of memory needed to ensure the limit is never reached [Bacon et al. 2003; Baker 1992; Jr. 1978]. More common are qualitative or experimental comparisons between GC schemes [Bacon et al. 2004; Grgic et al. 2018a].

Evaluating and Handling Collector Tradeoffs. Empirical issues with modern collectors are well known in the literature [Cai et al. 2022; Sareen and Blackburn 2022]. This paper complements that existing work with formal limits on how much collectors may improve. Huang et al. [2023] have attempted to help debug GC performance issues.

Dynamic Connectivity. Dynamic connectivity is a core problem in the study of dynamic graph algorithms. Many lower bounds are known [Larsen and Yu 2023; Pătraşcu and Demaine 2004] in the cell probe model [Yao 1978]. Nontrivial upper bounds are known for *undirected* dynamic connectivity, primarily using Euler tour trees [Holm et al. 2001; Kapron et al. 2013]. The contribution of our Section 7 is to show that, beyond just undirected graphs, *acyclic* directed graphs can also be handled efficiently for our variant of the problem. Section 4 proves that the lower bounds for the general connectivity problem also apply to the more restricted GC problem.

10 Conclusion

Collection delay is the worst-case time between a memory region first becoming unreachable in the heap and the GC collecting it. Modern real-time GC algorithms accept suboptimal collection delay in some pathological cases in order to guarantee constant-sized pause times when memory is abundant. This motivates asking whether modern GCs can be modified to reduce the worst-case delay without significantly increasing pause times. We provide a formal proof that extreme improvement in the worst case is not possible: superlogarithmic collection delay is inevitable for some pathological programs unless longer pause times are allowed. Our results hold for any GC implementing a formal mutator-observer style interface defined in this paper. Our proofs work via a nontrivial connection to fundamental data structures lower bounds, hence any stronger GC lower bound than ours would lead immediately to an improvement in important data structure lower bounds and vice-versa. We also describe a GC with some interesting asymptotic behavior, although it introduces too much overhead to recommend in non-pathological settings.

Acknowledgements

I would like to thank the anonymous reviewers, whose suggestions have dramatically improved the quality of the paper; Geoff Ramseyer, Alex Ozdemir, David K. Zhang, and Scott Kovach for extended discussions improving this work; as well as Dawson Engler, Zachary Yedidia, Akshay Srivatsan, members of the Stanford theory group, and attendees of the Stanford software lunch, who provided helpful insights, conversations, and proofreading.

Data Availability Statement

Implementations of different GCs in the Lua interpreter (as evaluated in Section 7) are available at https://doi.org/10.5281/zenodo.14942311 [Sotoudeh 2025].

Full Version With Appendices

The full version of this paper, with appendices, is available at https://doi.org/10.5281/zenodo.14948284.

Funding Statement

This work was generously funded via grants NSF DGE-1656518 and Stanford IOG Research Hub 281101-1-UDCPQ 298911.

References

2023. Programming Language and compiler Benchmarks. https://programming-language-benchmarks.vercel.app/

David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 285–298. https://doi.org/10.1145/604131.604155

David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 50–68. https://doi.org/10.1145/1028976.1028982

David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 207–235. https://doi.org/10.1007/3-540-45337-7_12

Henry G. Baker. 1992. The treadmill: real-time garbage collection without motion sickness. *ACM SIGPLAN Notices* 27, 3 (1992), 66–70. https://doi.org/10.1145/130854.130862

D. Beronic, N. Novosel, Branko Mihaljevic, and Aleksander Radovan. 2022. Assessing Contemporary Automated Memory Management in Java - Garbage First, Shenandoah, and Z Garbage Collectors Comparison. In *45th Jubilee International Convention on Information, Communication and Electronic Technology, MIPRO 2022, Opatija, Croatia, May 23-27, 2022*, Neven Vrcek, Marko Koricic, Vera Gradisnik, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Snjezana Babic, Vlado Sruk, Dejan Skvorc, Alan Jovic, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Tihomir Katulic, Juraj Petrovic, Tihana Galinac Grbac, and Benjamin Kusen (Eds.). IEEE, 1495–1500. https://doi.org/10.23919/MIPRO55190.2022.9803445

J. Bloch. 2017. *Effective Java*. Pearson Education. https://books.google.com/books?id=BIpDDwAAQBAJ

Hans-Juergen Boehm. 1993. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 197–206. https://doi.org/10.1145/155090.155109

David R. Brownbridge. 1985. Cyclic Reference Counting for Combinator Machines. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 273–288. https://doi.org/10.1007/3-540-15975-4_42

Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 46–57. https://doi.org/10.1109/ISPASS55109.2022.00005

Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. 2020. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*. ACM, 38–44. https://doi.org/10.1145/3394450.3397469

Brent Christian and Stuart Marks. 2021. JEP 421: Deprecate Finalization for Removal. https://openjdk.org/jeps/421.

Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 553–570. https://doi.org/10.1145/2509136.2509557

Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1021–1032. https://doi.org/10.1145/3510003.3510107

George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657. https://doi.org/10.1145/367487.367501

Herbert L. Gelernter, J. R. Hansen, and Carl L. Gerberich. 1960. A Fortran-Compiled List-Processing Language. *J. ACM* 7, 2 (1960), 87–101. https://doi.org/10.1145/321021.321022

Go. [n. d.]. A Guide to the Go Garbage Collector. https://tip.golang.org/doc/gc-guide.

Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association. https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog

H. Grgic, Branko Mihaljevic, and Aleksander Radovan. 2018a. Comparison of garbage collectors in Java programming language. In *41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018*, Karolj Skala, Marko Koricic, Tihana Galinac Grbac, Marina Cicin-Sain, Vlado Sruk, Slobodan Ribaric, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Predrag Pale, and Matej Janjic (Eds.). IEEE, 1539–1544. https://doi.org/10.23919/MIPRO.2018.8400277

H. Grgic, B. Mihaljević, and A. Radovan. 2018b. Comparison of garbage collectors in Java programming language. In *2018 41st International Convention on Information and Communication Tec logy, Electronics and Microelectronics (MIPRO)*. 1539–1544. https://doi.org/10.23919/MIPRO.2018.8400277

Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, Rocco A. Servedio and Ronitt Rubinfeld (Eds.). ACM, 21–30. https://doi.org/10.1145/2746539.2746609

Martin Hirzel, Amer Diwan, and Matthew Hertz. 2003. Connectivity-based garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 359–373. https://doi.org/10.1145/949305.949337

Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)* 48, 4 (2001), 723–760.

Claire Huang, Stephen M. Blackburn, and Zixian Cai. 2023. Improving Garbage Collection Observability with Performance Tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, Rodrigo Bruno and Eliot Moss (Eds.). ACM, 85–99. https://doi.org/10.1145/3617651.3622986

Roberto Ierusalimschy. [n. d.]. Garbage Collection in Lua. https://www.lua.org/wshop18/Ierusalimschy.pdf.

Richard E. Jones, Antony L. Hosking, and J. Eliot B. Moss. 2011. *The Garbage Collection Handbook: The art of automatic memory management*. CRC Press. http://gchandbook.org/

Henry G. Baker Jr. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (1978), 280–294. https://doi.org/10.1145/359460.359470

Jeehoon Kang and Jaehwang Jung. 2020. A marriage of pointer- and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 314–328. https://doi.org/10.1145/3385412.3385978

Bruce M. Kapron, Valerie King, and Ben Mountjoy. 2013. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, Sanjeev Khanna (Ed.). SIAM, 1131–1142. https://doi.org/10.1137/1.9781611973105.81

Hongjune Kim, Seonmyeong Bak, and Jaejin Lee. 2014. Lightweight and block-level concurrent sweeping for javascript garbage collection. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, Youtao Zhang and Prasad Kulkarni (Eds.). ACM, 155–164. https://doi.org/10.1145/2597809.2597824

Donald Ervin Knuth. 1997. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley. https://www.worldcat.org/oclc/312910844

Kasper Green Larsen and Huacheng Yu. 2023. Super-Logarithmic Lower Bounds for Dynamic Graph Problems. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*. IEEE, 1589–1604. https://doi.org/10.1109/FOCS57990.2023.00096

Henry Lieberman and Carl Hewitt. 1983. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (1983), 419–429. https://doi.org/10.1145/358141.358147

Rafael Dueire Lins. 1992. Cyclic Reference Counting with Lazy Mark-Scan. *Inf. Process. Lett.* 44, 4 (1992), 215–220. https://doi.org/10.1016/0020-0190(92)90088-D

Feng Mao, Eddy Z. Zhang, and Xipeng Shen. 2009. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009.* ACM, 91–100. https://doi.org/10.1145/1508293.1508307

Peter Marshall. [n. d.]. Trash talk: the Orinoco garbage collector. https://v8.dev/blog/trash-talk.

Alejandro D. Martínez, Rosita Wachenchauzer, and Rafael D. Lins. 1990. Cyclic reference counting with local mark-scan. *Inform. Process. Lett.* 34, 1 (1990), 31–35. https://doi.org/10.1016/0020-0190(90)90226-N

J. Harold McBeth. 1963. Letters to the editor: on the reference counter method. *Commun. ACM* 6, 9 (1963), 575. https://doi.org/10.1145/367593.367649

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. https://doi.org/10.1145/367177.367199

Marvin L Minsky. 1963. A LISP garbage collector algorithm using serial secondary storage. (1963).

Allen Newell and J. C. Shaw. 1957. Programming the logic theory machine. In *Papers presented at the 1957 western joint computer conference: Techniques for reliability, IRE-AIEE-ACM 1957 (Western), Los Angeles, California, USA, February 26-28, 1957*, Morton M. Astrahan (Ed.). ACM, 230–240. https://doi.org/10.1145/1455567.1455606

Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 349–365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen

Oracle. [n. d.]. Java Garbage Collection Basics. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.

Mihai Pătraşcu and Erik D. Demaine. 2004. Lower bounds for dynamic connectivity. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, László Babai (Ed.). ACM, 546–553. https://doi.org/10.1145/1007352.1007435

EJH Pepels, MCJD van Eekelen, and Marinus Jacobus Plasmeijer. 1988. *A cyclic reference counting algorithm and its proof.* Department of Theoretical Computer Science and Computational Models, Faculty ....

Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise garbage collection for C. In *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009*, Hillel Kolodner and Guy L. Steele Jr. (Eds.). ACM, 39–48. https://doi.org/10.1145/1542431.1542438

Pablo Galindo Salgado and Irit Katriel. [n. d.]. Garbage collector design. https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md.

J.D. Salkild. 1985. Implementation and analysis of two cyclic reference counting algorithms.

Kunal Sareen and Stephen Michael Blackburn. 2022. Better Understanding the Costs and Benefits of Automatic Memory Management. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, MPLR 2022, Brussels, Belgium, September 14-15, 2022*, Elisa Gonzalez Boix and Tobias Wrigstad (Eds.). ACM, 29–44. https://doi.org/10.1145/3546918.3546926

Matthew Sotoudeh. 2025. *(Artifact) Pathological Cases for a Class of Reachability-Based Garbage Collectors.* https://doi.org/10.5281/zenodo.14942312

Robert Endre Tarjan and Uzi Vishkin. 1984. Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time (Extended Summary). In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984.* IEEE Computer Society, 12–20. https://doi.org/10.1109/SFCS.1984.715896

Bill Venners. 1998. Object Finalization and Cleanup How to Design Classes for Proper Object Cleanup. https://www.artima.com/articles/object-finalization-and-cleanup.

Joseph Weizenbaum. 1962. Knotted list structures. *Commun. ACM* 5, 3 (1962), 161–165. https://doi.org/10.1145/366862.366897

Joseph Weizenbaum. 1964. More on the reference counter method of erasing list structures. *Commun. ACM* 7, 1 (1964), 38. https://doi.org/10.1145/363872.363881

Joseph Weizenbaum. 1969. Recovery of reentrant list structures in SLIP. *Commun. ACM* 12, 7 (1969), 370–372. https://doi.org/10.1145/363156.363159

Josh Wolfe. 2017. Why Zig When There is Already C++, D, and Rust? https://ziglang.org/learn/why_zig_rust_d_cpp/#no-hidden-allocations

Andrew Chi-Chih Yao. 1978. Should Tables Be Sorted? (Extended Abstract). In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978.* IEEE Computer Society, 22–27. https://doi.org/10.1109/SFCS.1978.33

A Extended Motivating Examples

Full code for the motivating examples from Section 2 is provided below. We also describe modified versions that would cause similarly pathological behavior on GCs supporting generational and/or reference counting extensions to their collectors.

A.1 GC Thrashing

The full Lua code is given below. We simulate large allocation blobs via descriptors, and keep track of the number of allocated blobs that have not yet been collected. This tracks logical memory, ignoring the overhead of the interpreter itself. This is likely what a user not interested in modifying the interpreter would end up implementing. It would also be reasonable to implement the blobs as allocations on a different machine, or reservations on some external resource (e.g., open network connections).

```
START_TIME = os.clock()

TOTAL_ALLOC = 0
N_ITEMS = 1000000

ALLOC_SIZE = 1
ALLOC_BUFFER = 1000

ALLOC_LIMIT = (ALLOC_SIZE * N_ITEMS) + ALLOC_BUFFER
-- arg[1] can be "limit", "nolimit", or "manual"
ENFORCE_LIMIT = (arg[1] ~= "nolimit")
MANUAL_FREE = (arg[1] == "manual")

local function clock(message)
    local time = os.clock()
    local lpad = 20
    if #message < lpad then
        message = string.rep(" ", lpad - #message) .. message
    end
    print(message, "@", time - START_TIME, "m", TOTAL_ALLOC)
    -- update the START_TIME to ignore time taken printing the clock message
    START_TIME = START_TIME + (os.clock() - time)
end

local function collect_blob(self)
    TOTAL_ALLOC = TOTAL_ALLOC - self.fake_alloc_size
end

local function fresh_blob()
    -- simulate an allocation
    if ENFORCE_LIMIT and TOTAL_ALLOC >= ALLOC_LIMIT then
        clock("Start collection")
        collectgarbage("collect")
        clock("End collection")
        if TOTAL_ALLOC >= ALLOC_LIMIT then
            print("OUT OF MEMORY!")
            os.exit(1)
        end
```

```
    end
    TOTAL_ALLOC = TOTAL_ALLOC + ALLOC_SIZE
    local res = setmetatable({fake_alloc_size=ALLOC_SIZE}, {__gc=collect_blob})
    return res
end

local function free_blob(blob)
    TOTAL_ALLOC = TOTAL_ALLOC - blob.fake_alloc_size
    blob.fake_alloc_size = 0
    setmetatable(blob, {})
end

local function fetch_item()
    return {data=fresh_blob()}
end

local function process_item(item)
    local work_space = fresh_blob()
    if MANUAL_FREE then
        free_blob(work_space)
    end
    return 1
end

-- stage 1: fetch a lot of items to process
local items = {}
for i=1,N_ITEMS do
    items[i] = fetch_item()
    clock("Iter")
end
clock("STAGE")
-- stage 2: process each item
local summary = 0
for i=1,N_ITEMS do
    summary = summary + process_item(items[i])
    clock("Iter")
end
clock("STAGE")
-- Don't bother GC'ing at the end
io.flush()
os.exit(0)
```

A.2 GC Thrashing: Breaking Reference Counting and Generations

The above code performs well under both generational and reference counting-based collectors. However, it can be easily modified to make reference counting useless on the example: simply make `fresh_blob()` create and return a node from a cyclicly linked list. To cause similar behavior on a language using generational collection, the second stage could swap the blob allocated in `process_item` with one of the blobs allocated in the first stage. Then it will be the older allocation region that becomes unreachable, hence generational collection will not help.

A.3 Delayed Finalization

The delayed finalization server and client scripts are given below.

```
------------- SERVER
N_SLOTS=100

local slot_open = {}
for i=1,N_SLOTS do slot_open[i] = true end

local worklist = {}

os.execute("mkdir -p work")
os.execute("rm -f work/*")

local function path_exists(path)
  local f, _, _ = io.open(path, "r")
  if f then
    f:close()
    return true
  end
  return false
end

local function finalize(descriptor)
  os.remove(descriptor.path)
  slot_open[descriptor.i] = true
end

local function poll()
  -- phase 1: poll for more work
  for i=1,N_SLOTS do
    if slot_open[i] then
      local fname = "work/" .. tostring(i) .. ".txt"
      if path_exists(fname) then
        slot_open[i] = false
        table.insert(worklist,setmetatable({i=i,path=fname},{__gc=finalize}))
      end
    end
  end
end

local function process(item)
    print("Processing ... " .. tostring(item.i))
    item = nil
end

local function drain()
  -- phase 2: drain the work queue
  while #worklist > 0 do
    process(table.remove(worklist))
  end
end
```

```
while true do
  poll()
  drain()
end

------------- CLIENT (different file)
-- poll, waiting for our slot to open up. when it does, fill it.

TARGET=10000
MY_SLOT=1

local count = 0
while count < TARGET do
  local fname = "work/" .. tostring(MY_SLOT) .. ".txt"
  local f, _, _ = io.open(fname, "r")
  if not f then
    local f = assert(io.open("work/tmp.txt", "w"))
    f:write("Hello, World!\n")
    f:flush()
    f:close()
    os.rename("work/tmp.txt", fname)
    count = count + 1
    print(count)
  else
    f:close()
  end
end
```

A.4 Delayed Finalization: Breaking Reference Counting and Generations

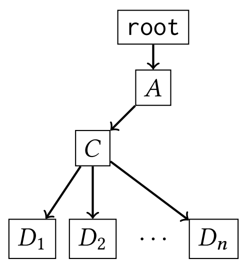
Here again, reference counting would not help if each workitem were made part of a cyclic list. Generational collection would also fail to help if, e.g., it takes enough time to process workitems that they become part of an older generation.

B Counterexample to Claim of Pepels et. al.

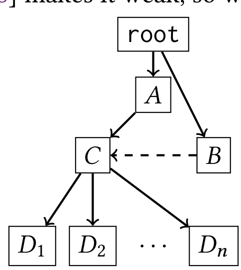
Pepels et al. [1988] claims that their cyclic reference counting scheme "has no time overhead compared to classical reference counting algorithms, if there are no cycles in the graph." This claim would directly contradict our lower bound (Theorem 5.4). Unfortunately, no extended justification for the claim is given in the text. We believe that the claim is a mistake in the English translation.

We now present a sequence of operations that causes their algorithm to take asymptotically more time than reference counting, even though there are no cycles in the graph, hence directly refuting their claim.

First, construct the following heap, noticing that all pointers must be assigned strong because the heap is treeshaped:



Now, allocate a new memory region B and copy the $A \to C$ pointer to $B \to C$. Notice that the `CopyPtr` method in Pepels et al. [1988] makes it weak, so we have the following heap:



Finally, call `DeletePtr` on the edge $A \to C$. Following the definition of the `DeletePtr` method in Pepels et al. [1988], it, among other things:

(1) Calls `AdjustPresumableCycle` on C, which:
  (a) Calls `Swap` on C, making $B \to C$ strong,
  (b) Calls `AdjustGraphBetween` on C, C, which:
    (i) Iterates over the strong-pointer sons of C.

But at that point, C has n strong-pointer sons, namely D_1 through D_n. Hence the `DeletePtr` operation will take $O(n)$ time, even though the heap has always been acyclic and reference counting would have only taken $O(1)$ time.

Algorithm 24: `DCInsert`$(v_i \leftrightarrow v_j)$

Rename i, j so $L(v_i) \leq L(v_j)$;
$E(v_i \leftrightarrow v_j) \leftarrow E(v_i \leftrightarrow v_j) + 1$;
use_lpcds $\leftarrow$ use_lpcds $\wedge L(v_i) = L(v_j) - 1$;
use_lpcds $\leftarrow$ use_lpcds $\wedge$ indeg$(v_j) \leq 1$;
use_lpcds $\leftarrow$ use_lpcds $\wedge$ outdeg$(v_i) \leq 1$;
if use_lpcds **then**
  `LPCInsert`$(v_i \rightarrow v_j)$;

Algorithm 25: `DCInit()`

$E, L,$ use_lpcds$, R \leftarrow \emptyset, \emptyset, ?, \emptyset$;

Algorithm 26: `DCDelete`$(v_i \leftrightarrow v_j)$

Rename i, j so $L(v_i) \leq L(v_j)$;
$E(v_i \leftrightarrow v_j) \leftarrow E(v_i \leftrightarrow v_j) + 1$;
if use_lpcds **then**
  `LPCDelete`$(v_i \rightarrow v_j)$;

Algorithm 27: `DCConnected`$(v_i \leftrightarrow^+ v_j)$

$R \leftarrow R \cup \{v_i\}$;
if use_lpcds $=? \wedge |R| = n$ **then**
  if *DFS from R finds graph is n disjoint paths* **then**
    forall *vertex v* **do**
      $L(v) \leftarrow$ distance of v from a node in R;
    Initialize the LPCDS with layers from L and edges from E;
    use_lpcds $\leftarrow \top$;
  else
    use_lpcds $\leftarrow \bot$;
if use_lpcds $= \top \wedge v_i \in R$ **then** **return** `LPCConnected`$(v_i \rightarrow^+ v_j)$;
return *BFS or DFS connectedness search*;

Fig. 10. Showing our directed LPCDS can be used to speed up a general DCDS when the graph is of the form used by Pătraşcu and Demaine [2004].

C Layered Permutation Graphs

Section 5.1 formalized the LPCDS slightly differently from the original in Pătraşcu and Demaine [2004]. They phrase their lower bound in terms of the general undirected DCDS problem, but the sequence of operations that they prove requires $\Omega(\log n)$ time satisfies the conditions of the LPCDS we defined (Definition 5.1). In particular, their sequence of operations always involves permuting one layer of edges at a time, then querying for reachability from each node in the input layer to some node in a later layer.

To show that lower bounds from Pătraşcu and Demaine [2004] apply to the LPCDS problem we defined in Definition 5.1, it suffices to show that an efficient LPCDS could be used to construct an undirected DCDS that has identical amortized time complexity to the LPCDS when the queries are in the form used by Pătraşcu and Demaine [2004], i.e., supported by the LPCDS. Such a reduction is shown in Figure 10.

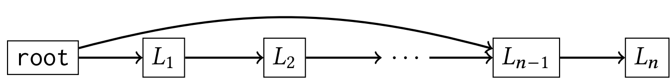
The basic idea is to wrap the LPCDS in another data structure. As we get graph updates, we store them in a naïve graph representation. When the first connectedness query is made, we perform a DFS to determine if the graph is of the LPCDS form; if so, we assign each node its proper layer number, initialize the LPCDS, and then proceed by querying the LPCDS. If the LPCDS invariants are ever violated, we go back to the naïve graph algorithm.

D Pathological Example for Baker's Treadmill

Baker's treadmill algorithm [Baker 1992; Jr. 1978] is a paradigmatic example of a real-time garbage collector. All regions except `root` are initially marked white; `root` is marked gray. During each collector operation, some constant number of gray node(s) are turned black and their white children are made gray. When all nodes are either black or white, i.e., none are gray, we know the white nodes are unreachable and we may free them. Finally, the black nodes are marked as white, `root` is marked gray, and the process repeats.

Baker [Baker 1992; Jr. 1978] proves that *if the program satisfies a number of conditions* regarding, e.g., how frequently memory is allocated vs. released, then the collector will always make enough progress to ensure at least one free memory region is known by the allocator whenever the program requests memory. Unfortunately, there exist programs that force Baker's treadmill to either introduce superconstant pause times, i.e., breaking the real-time guarantee, or incorrectly report that there is no available memory. This section briefly describes one such program.

The program first fills up the available memory with a single large linked list, preserving a pointer to the second-to-last node in the list:



Assume that after building this heap the collector completes a flip, i.e., one complete pass of the collector through the heap. This can be triggered either by requesting a new memory region or by performing no-ops until the mark-and-sweep process is completed.

In any case, the heap now has `root` marked in gray and all other nodes in white. Suppose the program now deletes the $L_{n-1} \rightarrow L_n$ pointer. This makes L_n unreachable, but the collector is unable to guarantee that until it finishes marking the *entire* rest of the heap $L_1, \ldots, L_{n-1}$. Hence, if a new allocation is requested immediately after deleting the edge to L_n, the collector will be forced to either complete the $O(n)$-time mark-and-sweep pass or report that it cannot find a free region, even though L_n is indeed unreachable. This scenario shows how pathological programs can cause non-real-time behavior if they operate too close to the memory limit.

Note in this particular scenario, combining the treadmill with a reference counting collector would enable quick collection. However, our lower bounds guarantee pathological examples will always exist: e.g., L_n could be the head of a cycle in the graph, in which case naïve reference counting would not help.

E Lua Experiment Details

We forked the standard Lua 5.4.6 interpreter, removing its garbage collection code and inserting hooks where Lua VM instructions manipulate Lua references. These hooks call `GCAllocate`, `GCInsert`, and `GCDelete` methods of the underlying GCDS. We wrote two GCDS implementations, one for our GCDS and the other for SynCC (prior work [Bacon and Rajan 2001]).

Implementation of our GCDS. Our GCDS implementation is based on the algorithm in Section 7. We used splay trees to implement the Euler tour trees.

Implementation of SynCC. We compared to the SynCC approach of Bacon and Rajan [2001]. Later in their paper Bacon and Rajan [2001] describe how it can be used in a delayed fashion but we implemented it as it is first presented, i.e., a collector with $d(n) = 1$, to compare against our GCDS. `GCAllocate`, `GCInsert`, and `GCDelete` maintain a copy of the points-to graph. After deletion of an edge, we apply the algorithm in Bacon and Rajan [2001] to determine whether the region is reachable or, if not, to identify all regions that can be freed.

Algorithm 28: GCInsert($v_i \to v_j$)

DCInsert($v_i \to v_j$);
$E(v_i \to v_j) \leftarrow E(v_i \to v_j) + 1$;

Algorithm 29: GCAllocate()

$v_i \leftarrow$ fresh node;
DCInsert(root $\to v_i$);
return v_i;

Algorithm 30: GCDelete($v_i \to v_j$)

DCDelete($v_i \to v_j$);
if ¬DCConnected(root $\to^+ v_j$) **then**
    $D \leftarrow \{v_j\}$;
    forall $E(v_j \to m) > 0$ **do**
        $D \leftarrow D \cup$ GCDelete($v_j \to m$);
    FreeList $\leftarrow$ FreeList $\cup D$;

Algorithm 31: GCStep()

no-op;

Fig. 11. Reduction from GCDS to DCDS.

Memory Overhead. For both implementations we made an effort to reduce memory overhead where obvious options existed, e.g., using bitfields for boolean flags, but focused on implementation simplicity and time overhead first and foremost. In particular, for implementation simplicity we maintained a copy of the points-to graph in a separate graph data structure. The default Lua collector embeds this graph within the elements themselves, which allows it to achieve better memory overhead. Unfortunately, that approach requires type-specific iterators, hence complicating implementation of the GC, hence our simpler but higher-memory-usage approach.

General Benchmarks Used. All of the general benchmarks (Section 7.6.2) except for `list-4096` and `dbllist-4096` are from the Programming Languages Benchmarks project [plb 2023]. We excluded the `coro-prime-sieve` benchmark, as it caused a stack overflow even on the unmodified Lua interpreter when run on inputs large enough to produce nontrivial time. The `list-4096` benchmark creates a large linked list, iterates over it to compute its length, and then deletes it. The `dbllist-4096` benchmark is identical, except it builds a doubly linked list.

F Reduction from GCDS to DCDS

Most of the paper proved that an efficient GCDS could be used to build an efficient DCDS; this section proves the converse. The result of this proof is to show that discovering improved bounds on the GCDS problem is just as hard as discovering improved bounds on the DCDS problem.

Theorem F.1. *Assume the existence of an $O(g(n))$-time DCDS. Then there exists a GCDS with $d(n) = 1$ and $t_{\mathrm{AR}}(n) = O(g(n))$.*

Proof. The reduction is shown in Figure 11. When a pointer is deleted we use the DCDS as an oracle to check whether there remains any path from `root` to the node; if not, we delete it and recurse on its outgoing pointers similar to reference counting. □

G GC to GCDS

Having a formal model is unavoidable when proving rigorous bounds; to prove bounds on ‘garbage collection’ we must clearly delineate what behaviors we require of ‘garbage collection.’ We designed the GCDS interface in Definition 3.1 to match the standard mutator-observer interface provided by existing inplace GCs for imperative programs. It is interesting to ask whether there might be an imperative programming language using a GC that can *not* be made to fit this interface without performance degradation. With some caveats (below), the answer is no.

Given access to a general-purpose, imperative programming language like Python or Lua with a hypothetical efficient GC, you could simulate a GCDS by making `GCAllocate` create a new set object for the memory region and simulating `GCInsert`$(a \to b)$ and `GCDelete`$(a \to b)$ by adding and removing references to b from the set in memory region a. Memory regions can be annotated with finalizers that add them to `GCFreeList`.

Caveat 1: Need for Finalizers. The sketch above requires the language support finalizers so that the simulated GCDS can determine when a region is collectable, i.e., can be added to `GCFreeList`. Some sort of feedback from the GC to the application seems necessary, as otherwise the GC could simply never free anything. Perhaps the simplest GC interface would require the ability to set a memory limit and then query whether a new allocation can be made or not. A language supporting such a GC could be used to implement a GCDS variant where instead of a free list the GCDS allows querying only whether or not anything has been collected. This GCDS variant would be sufficient to carry out our reductions in Section 4 and Section 5, so our results still hold in this setting.

Caveat 2: Nonlocal Modifications. The final caveat is that the GCDS allows operations involving any nodes reachable from `root`, but in an imperative program, each operation can usually only write to a region reachable within a constant number of pointer indirections from a local or global variable. This corresponds to adding the additional restriction in Definition 3.1 that all operations must involve nodes at most some bounded distance from `root`. Thankfully, our reduction in Section 4 satisfies this requirement, and our reduction in Section 5 can be made to satisfy this requirement by introducing an auxiliary node similar to the X node used in Section 4. Hence, our results still hold in this setting as well.

H Rigorous Definitions for GC Performance

In this section we give more rigorous definitions for the notions of GCDS delay and running time sketched in Section 3.2. We will assume here a deterministic GCDS; see Section I for discussion of nondeterministic GCDS. The first key idea is the *associated heap multigraph* of a sequence of GCDS operations, which associates with each sequence of GCDS operations the state of the heap that those operations describes. Notably, the GCDS is not required to actually store the associated heap multigraph (although it is frequently helpful, see Section 3.3; rather, the associated heap multigraph is a theoretical construction that we use in our definitions of GCDS performance.

Definition H.1. Let $p_1, \ldots, p_n$ be a sequence of GCDS operations (Definition 3.1) and $r_1, \ldots, r_n$ be the values returned by the GCDS for each of those operations in the sequence (or $\perp$ if the operation returns nothing). Then, the *associated heap multigraph* $\eta(p_1, \ldots, p_n)$ is a directed multigraph (i.e., directed graph with possibly multiple edges between the same pair of nodes) defined inductively:

(1) If $n = 0$, then $\eta(\emptyset)$ is a graph with a single node, named `root`, and no edges.
(2) If p_n is `GCAllocate()`, then $\eta(p_1, \ldots, p_n)$ is $\eta(p_1, \ldots, p_{n-1})$ with a new node r_n added to it, along with a new edge `root` $\to r_n$.
(3) If p_n is `GCInsert`$(a \to b)$, then $\eta(p_1, \ldots, p_n)$ is $\eta(p_1, \ldots, p_{n-1})$ with a new edge $a \to b$ (if a and b are nodes in the graph).
(4) If p_n is `GCDelete`$(a \to b)$, then $\eta(p_1, \ldots, p_n)$ is $\eta(p_1, \ldots, p_{n-1})$ with one edge $a \to b$ removed (if any exist).
(5) If p_n is `GCStep()`, then $\eta(p_1, \ldots, p_n)$ is $\eta(p_1, \ldots, p_{n-1})$.

Recall from Definition 3.1 that a GCDS could make some reasonable assumptions about its inputs, e.g., that no node is used unless it is reachable from `root`. We now define precisely the meaning of a sequence being *valid*. In particular, GCDS behavior need not be defined on invalid sequences.

Definition H.2. Let $p_1, \ldots, p_n$ be a sequence of GCDS operations (Definition 3.1). The sequence is *valid* if it satisfies the following inductive definition.

(1) If $n = 0$, then the sequence is valid.
(2) If p_n is `GCAllocate()` or `GCStep()`, then $p_1, \ldots, p_n$ is valid if $p_1, \ldots, p_{n-1}$ is valid.
(3) If p_n is `GCInsert`$(a \to b)$, then $p_1, \ldots, p_n$ is valid if (1) $\eta(p_1, \ldots, p_{n-1})$ is valid, and (2) `root` can reach a and b in $\eta(p_1, \ldots, p_{n-1})$.
(4) If p_n is `GCDelete`$(a \to b)$, then $p_1, \ldots, p_n$ is valid if (1) $\eta(p_1, \ldots, p_{n-1})$ is valid, (2) `root` can reach a and b in $\eta(p_1, \ldots, p_{n-1})$, and (3) there is at least one edge $a \to b$ in $\eta(p_1, \ldots, p_{n-1})$.

We can now define more rigorously the *worst-case delay* $d(n)$, which in Definition 3.2 we described as "the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes (i.e., making at most n calls to `GCAllocate`), if some operation makes a node u unreachable, u is added to `GCFreeList` after at most $d(n)$ more GCDS operations (including the operation that makes it unreachable)."

Definition H.3. Let S_n be the set of valid nonempty GCDS sequences $p_1, \ldots, p_k$ such that at most n of the p_is are `GCAllocate()`s. Consider some particular GCDS, and let $F(p_1, \ldots, p_i)$ be the set of all nodes added by the GCDS to `GCFreeList` during the sequence of operations $p_1, \ldots, p_i$. Then the *worst-case delay* $d(n)$ of the GCDS is defined to be:

$$\begin{aligned} d(n) := \max(\{\delta \quad | \quad & \text{exist } p_1, \ldots, p_k \in S_n,\ t \leq k - \delta + 1, \text{ and } a \in \eta(p_1, \ldots, p_{t-1}) \text{ such that} \\ & (1) \text{ there is a path } \texttt{root} \to^+ a \text{ in } \eta(p_1, \ldots, p_{t-1}), \\ & (2) \text{ there is not a path } \texttt{root} \to^+ a \text{ in } \eta(p_1, \ldots, p_t), \text{ and} \\ & (3)\ a \notin F(p_1, \ldots, p_{t-1+\delta})\}). \end{aligned}$$

If there is no maximum, we write $d(n) = \infty$.

Intuitively, this definition is looking at sequences $p_1, \ldots, p_k$ and positions t in that sequence such that a became unreachable on operation p_t, and a has not been added to the free list by operation $p_{t-1+\delta}$. The largest such δ, over all sequences involving at most n nodes, is called $d(n)$.

We can also define the *worst-case pause time* of a GCDS, which we earlier described as "the pointwise minimal function such that, in any sequence of GCDS operations involving at most n nodes, each GCDS operation in the sequence takes time at most $t(n)$."

Definition H.4. Let S_n be the set of valid nonempty GCDS sequences $p_1, \ldots, p_k$ such that at most n of the p_is are `GCAllocate()`s. Consider some particular GCDS and let $T(p_1, \ldots, p_k)$ be the time taken by the GCDS to process the valid sequence of operations $p_1, \ldots, p_k$; in particular, $T(p_1, \ldots, p_k) - T(p_1, \ldots, p_{k-1})$ is the marginal time to process p_k. Then the *worst-case pause time* is

$$t(n) := \max(\{T(p_1, \ldots, p_k) - T(p_1, \ldots, p_{k-1}) \mid p_1, \ldots, p_k \in S_n\}).$$

Note in particular that S_n is closed under taking (nonempty) prefixes, so the above definition also captures the time taken for every subsequence of operations.

We can now define t_{AR}, $t_{\text{A,AR}}$, and $t_{\text{S,A,AR}}$. Each one is very similar to the definition of worst-case pause time, except taking the maximum over an increasing small subset of sequences.

Definition H.5. Using the definitions from Definition H.4, the *all-reachable* worst-case pause time $t_{\text{AR}}(n)$ of a GCDS is restricted to sequences that never make any node unreachable:

$$\begin{aligned} t_{\text{AR}}(n) := \max(\{T(p_1, \ldots, p_k) - T(p_1, \ldots, p_{k-1}) \mid & \text{ exists } p_1, \ldots, p_k \in S_n \text{ such that, for every } i \leq k, \\ & (1) \text{ every node in } \eta(p_1, \ldots, p_i) \text{ is reachable from } \texttt{root}\}). \end{aligned}$$

Definition H.6. Using the definitions from Definition H.4, the *acyclic all-reachable* worst-case pause time $t_{A,AR}(n)$ of a GCDS is restricted to sequences that never make any node unreachable and never introduce a cycle:

$$t_{A,AR}(n) \coloneqq \max(\{T(p_1,\ldots,p_k) - T(p_1,\ldots,p_{k-1}) \mid \text{exists } p_1,\ldots,p_k \in S_n \text{ such that, for every } i \le k, \\ \text{(1) every node in } \eta(p_1,\ldots,p_i) \text{ is reachable from } \texttt{root}\text{, and} \\ \text{(2) the graph } \eta(p_1,\ldots,p_i) \text{ is acyclic}\}).$$

The definition of sparse acyclic all-reachable worst-case pause time is fundamentally asymptotic due to the definition of sparsity; we first define a corresponding nonasymptotic notion of *m-sparse* acyclic all-reachable worst-case pause time.

Definition H.7. Using the definitions from Definition H.4, the *m-sparse acyclic all-reachable* worst-case pause time $t_{mS,A,AR}(n)$ is restricted to sequences that never make any node unreachable, never introduce a cycle, and never have more than m edges leaving any node.

$$t_{mS,A,AR}(n) \coloneqq \max(\{T(p_1,\ldots,p_k) - T(p_1,\ldots,p_{k-1}) \mid \text{exists } p_1,\ldots,p_k \in S_n \text{ such that, for every } i \le k, \\ \text{(1) every node in } \eta(p_1,\ldots,p_i) \text{ is reachable from } \texttt{root}\text{, and} \\ \text{(2) the graph } \eta(p_1,\ldots,p_i) \text{ is acyclic, and} \\ \text{(3) all nodes in } \eta(p_1,\ldots,p_i) \text{ have outdegree at most } m\}).$$

Finally, $t_{S,A,AR}$ is defined as the asymptotic growth rate of the $t_{mS,A,AR}$s, if an upper bound on the growth rate exists independent of m (otherwise we write $t_{S,A,AR}(n) = \infty$.

Definition H.8. A GCDS has *sparse acyclic all-reachable* worst-case pause time $t_{S,A,AR}(n)$ if for every constant m, $t_{mS,A,AR}(n) = O(t_{S,A,AR}(n))$.

I Variants of Definitions Our Bounds Work For

We now briefly discuss two alternate definitions that our bounds also work for, and how using these alternate definitions makes the bounds apply to other GCDS implementations.

I.1 First-Delay

Above, we defined delay to be the maximum number of operations between when a node becomes unreachable and when it is added to the free list. But our reductions actually only rely on the ability to detect when *anything* has been made unreachable, i.e., we only really need to look at the difference between *the first node becoming unreachable* in a sequence and when any node gets added to the free list. This leads to the following definition of *first delay*.

Definition I.1. Let S_n be the set of valid nonempty GCDS sequences $p_1,\ldots,p_k$ such that at most n of the p_is are `GCAllocate()`s. Consider some particular GCDS, and let $F(p_1,\ldots,p_i)$ be the set of all nodes added by the GCDS to `GCFreeList` during the sequence of operations $p_1,\ldots,p_i$. Then the *worst-case first delay* $d_1(n)$ of the GCDS is defined to be:

$$d_1(n) \coloneqq \max(\{\delta \mid \text{exists } p_1,\ldots,p_k \in S_n \text{ and } t \le k-\delta \text{ such that} \\ \text{(1) } \texttt{root} \text{ can reach every node in } \eta(p_1,\ldots,p_{t-1}), \\ \text{(2) there is some node that } \texttt{root} \text{ cannot reach in } \eta(p_1,\ldots,p_t)\text{, and} \\ \text{(3) } F(p_1,\ldots,p_{t-1+\delta}) \text{ is empty}\}).$$

The results in Section 4 and Section 5 all still hold when worst-case delay is replaced with worst-case first delay.

This observation is important because it ensures our results still apply to schemes that do lazy traversal of the unreachable space [Weizenbaum 1969]. For example, consider a GC guaranteeing that whenever there are unreachable regions, at least *one* of them is on the free list, but not necessarily all — some unreachable regions may be missing from the free list until it becomes empty, at which case the GC goes searching for another unreachable element to add.[2] Using our normal notion of delay, any lazy scheme meeting that guarantee has $d(n) = \infty$ because, unless more space is requested, nodes after the head might never be added to the free list. But this does not really capture the goal behind delay, because such a scheme might still ensure that all allocations can be serviced. However, this scheme would have $d_1(n) = 1$ constant *first delay*, because it guarantees that *something* is added to the free list immediately once *anything* first becomes unreachable.

In summary, our results actually imply that there are pathological programs for which no GC (that can be adapted to implement the GCDS interface) can quickly detect even when *the very first unreachable region* has become unreachable. Hence, rephrasing our results in terms of first delay makes them stronger, and shows how our lower bounds apply even to schemes using lazy reclamation techniques.

I.2 Randomized GCDS

So far in this paper, we have assumed deterministic algorithms. While it is not a major focus of our results, we now briefly discuss the question of whether and to what extent our lower bounds apply also to randomized algorithms.

First, we consider algorithms that make random decisions during their execution but guarantee correctness regardless of the random decisions used (i.e., the randomness can only affect performance). Our bounds apply without modification if the delay and pause times are defined for the worst-case random choices. Surprisingly, they also apply without modification when the delay and pause times are defined for the *best* possible random choice you could make with only the information read so far from the persistent store; this is because the lower bounds we reduce against [Larsen and Yu 2023; Pătraşcu and Demaine 2004] are proved in the cell probe model [Yao 1978], i.e., they only care about the sequence of reads from and writes to the persistent store.

The question becomes more complicated if the algorithm is only required to be correct with a certain (high) probability over the random decisions, e.g., under some very unlikely choices it is allowed to leave certain regions uncollected or collect regions that are still reachable. To the best of our knowledge neither of the bounds we reduce against work in this setting, and we are not aware of a way to extend our results to this setting.

J Motivating Issue 2: Predictability of Finalizers

Many programming languages allow the use of finalizers, which are functions attached to memory regions that get run right before the region is collected. One program that might seem reasonable to a programmer not expecting the complexity of modern GCs is excerpted in Figure 12. It is a server that polls for files in a directory and processes each one. The file descriptors are associated with a finalizer that deletes the underlying file when the descriptor is collected. A similar client program (not shown) inserts a file then waits for the file to be deleted (indicating the server has finished processing it) before adding another. In this way, the file acts as a lock, communicating to the client when the server is ready for more work.

[2] Weizenbaum [1969] provides an example of such a scheme, except that it uses reference counting so it fails to meet even that guarantee in the presence of cycles (the basic idea is to delay recursive decrementing of child reference counts until the parent is actually allocated again).

```
-- (excerpted implementations away of some functions, globals)
local function finalize(descriptor)
  os.remove(descriptor.path)
  slot_open[descriptor.i] = true end
while true do
  -- phase 1: poll for more work
  for i=1,N_SLOTS do
    if slot_open[i] then
      local fname = "file_test/" .. tostring(i) .. ".txt"
      if path_exists(fname) then
        slot_open[i] = false
        table.insert(worklist,
            setmetatable({i=i,path=fname},{__gc=finalize})) end end end
  -- phase 2: drain the work queue
  while #worklist > 0 do process(table.remove(worklist)) end end
```

Fig. 12. Excerpt from the example server program showing deadlock caused by delayed finalization. The server shown here polls the filesystem looking for new work. Each workitem is associated with a finalizer that deletes the corresponding file when that workitem becomes unreachable. Files in the directory indicate open workitems, so the client program (not shown) treats the files as a lock, i.e., waits for files to be deleted from that directory before adding more work for this server. Unfortunately, Lua's GC heuristics *never* trigger collection for this program, resulting in a deadlock.

Unfortunately, collection delay means that, even when the file descriptor becomes unreachable, it may not be collected until much later in the program execution. Since unlocking is tied to collection of the descriptor, this can unnecessarily increase lock contention.

Even worse, many popular languages use heuristics that improve GC performance in typical cases but can break even the $O(n)$ delay guarantee in the worst case. Lua is one such language, and we were surprised to see this cause an immediate deadlock with this system because it performs too few memory operations to trigger garbage collection. In this case, the file descriptors from the first work iteration are never actually finalized, hence the underlying files are never removed, so the client is never able to add additional work, triggering deadlock.

This scenario is made worse by the fact that the GC is unpredictable. When the number of work slots is set to 5 collection gets triggered frequently enough to avoid deadlock. But if the number of slots set to either, e.g., 1 or 100, deadlock is encountered. Similarly, if the code processing each workitem method performs a lot of memory operations the GC might be triggered frequently, avoiding the deadlock. But, if the `process` method is optimized to perform fewer memory operations, the deadlock can appear unexpectedly. In a language like Python that supports both reference counting and infrequent garbage collection, the finalizers might be promptly called for every version of the program until the programmer introduces a cyclic data structure, triggering delayed collection and hence either deadlock or increased lock contention.

It is relatively easy to warn programmers of these issues, and dissuade them from attempting to use finalizers as a convenient method to detect when regions are no longer used by the program. However, finalizers are a useful language feature, and it is interesting to ask, as we do in this paper, whether there is any efficient way to make them significantly more predictable. Unfortunately, our impossibility result proves that this sort of scenario is impossible to avoid.

J.1 Implication of Main Lower Bound for Guaranteed Finalization

Suppose the language attempts to guarantee for the user that finalizers are called reliably and promptly, i.e., $d(n) = O(1)$. Such a guarantee would be convenient, and allow users to rely on finalization for locks, etc., while avoiding the deadlock described above.

Unfortunately, our lower bound in Section 4 implies that there must exist a program, even a program where nothing ever becomes unreachable, where this language introduces an $\tilde{\Omega}(\log^{3/2} n)$-length pause time after some program operation. Hence, no collector guaranteeing prompt finalization is suited for all real-time settings.

J.2 Delayed Finalization With Immediate GCs

We tried running the same motivating example using the immediate GCs described in Section 7.6. Both our approach and SynCC guarantee immediate finalization, hence avoiding the deadlock issue. They take approximately 1.8 and 2 seconds, respectively, to complete the benchmark, which is comparable to a manually managed version on the unmodified Lua interpreter that takes approximately 1.8 seconds. This again highlights the importance of reducing collection delay in GCs.

Received 2024-10-16; accepted 2025-02-18