TECHNICAL ASSESSMENT TASK

PROBLEM STATEMENT

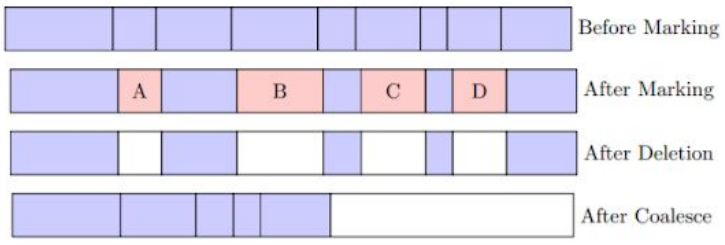
Compare Cheney's algorithm and Mark-compact algorithm with C++ and demonstrate which is better and why?. A C++ code demonstrating the principles of both this algorithm is expected.

Mark - Compact Algorithm

In computer science, a **mark-compact algorithm** is a type of garbage collection algorithm used to reclaim unreachable memory. Mark-compact algorithms can be regarded as a combination of the mark-sweep algorithm and Cheney's copying algorithm. First, reachable objects are marked, then a compacting step relocates the reachable (marked) objects towards the beginning of the heap area. This address the fragmentation caused by mark-sweep, which leads to significantly more efficient future allocations via the use of a "bump" allocator (similar to how a stack works), but adds on extra time and processing while GC is running because of the extra iteration(s).

Mark and Sweep

The earliest and most basic garbage collection algorithm is Mark-Sweep garbage collection - or a copy collector , and most modern algorithms are a variant on it. Mark-Sweep is a "stop-the world" collector, which means that at some point when the program requests memory and none is available, the program is stopped and a full garbage collection is performed to free up space.



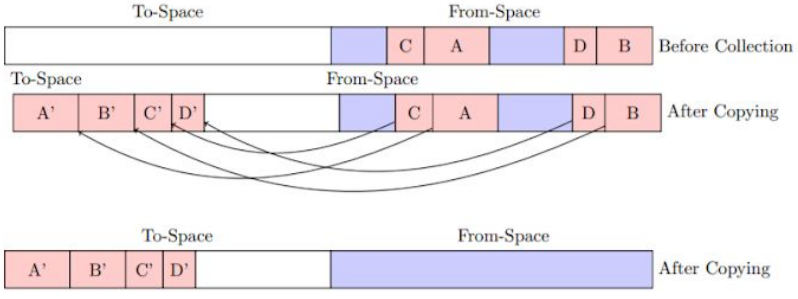
In the Mark-Sweep algorithm, each object has a "mark-bit" which is used during the collection process to track whether the object has been visited. Here is an algorithm for Mark-Sweep garbage collection implemented on top of some underlying explicit memory management routines, in which free regions of the heap are also considered objects with mark bits and a known size. According to McCarthy[5], the "stop-the-world" label was an effective description of the process involved with this garbage collector. The efficiency of a GC system depends primarily upon not

reaching the exhaustion point of available memory - the reason being that the memory reclamation process, according to McCarthy, required several seconds and the addition of several thousand registers to the free-list.

```
mark_sweep_collect() =
mark(root)
sweep()
mark(o) =
If mark-bit(o)=0
mark-bit(o)=1
For p in references(o)
mark(p)
EndFor
EndIf sweep() o = 0
```

```
sweep()
o = 0
While o < N
If mark-bit(o) = 1
mark-bit(o) = 0
Else
free(o)
ElseIf
o = o + size(o)
EndWhile
```

Cheney's Copy Algorithm



Cheney's Algorithm is a subset of garbage collectors known as copying collectors. In copying collectors, reachable objects are relocated from one address to another during a collection. Available memory is divided into two equal-size regions called the from-space and the to-space. During allocation, the system keeps a pointer into the to-space which is incremented by the amount of memory requested for each allocation. When there is insufficient space in to-space to fulfill an allocation, a collection is performed. A collection consists of swapping the roles of the regions, and copying the live objects from from-space to to-space, leaving a block of free space (corresponding to the memory used by all unreachable objects) at the end of the to-space. Since objects are moved during a collection, the addresses of all references must be updated. This is done by storing a "forwarding-address" for an object when it is copied out of-from-space. Like the mark-bit, this forwarding-address can be thought of as an additional field of the object, but is usually implemented by temporarily repurposing some space from the object.

Comparison between both the algorithms:

Cheney's algorithm is better than Mark- sweep algorithm

and it is explained why:

Complexity:

We assert asymptotic behavior is not an effective metric when used to compare the efficiency of garbage collection algorithms.Comparing the complexity of the Mark-Sweep and Copy Collector garbage collectors defined in section 3. We generalize the phases of the Mark-Sweep again below:

1. Initialize traces by clearing marked bits
2. Trace
3. Mark
4. Sweep
5. Reclaim allocated memory

We can say that phase 1 has a negligible effect on memory and time for both garbage collectors, thus we examine the complexity of both algorithm's with respect to phases 2, 3, 4, and 5.

When examining Cheney's Copy Collector, it is important to note that the algorithm forgoes direct allocation and instead swaps objects in regions of memory. Phase 3 of the Mark-Sweep algorithm can also be disregarded in the context of a Copy Collector. The sweep phase is linearly proportional to the number of active objects in the heap for both collectors. Regarding phase 4, since Cheney's algorithm in particular passes through the heap breadth first, we can say the complexity is linearly proportional to the number of active cells - a potential significant difference to the total size of the heap. In phase 5, we say that the time complexity is proportional to the size of the uncopied objects in the heap. Thus, we can conclude that asymptotic complexity of performing collection on a heap of n objects is proportional to n regardless of the algorithm used.

According to Hertz, despite having worse worst-case space performance, for realistic applications, the Mark-Sweep algorithm is preferable in terms of space, but the copying collection offers much faster allocation than the traditional sweep and allocate technique used by Mark-Sweep.

Despite stating the equivalence of asymptotic complexity, we can conclude that for most practical applications involving a lisp-like language Cheney's Copy Collector surpasses a traditional Mark-Sweep GC in terms of speed and worst-case space performance.

Allocation Timing:

Cheney's algorithm has a key advantage over non-copying techniques for allocation timing because it does not require the use of a complete allocator, with freeing and coalescing. Because the entire heap is copied over and freed all at once, it simply needs to use a "bump pointer" for allocation: the algorithm simply tracks the end of

the allocated heap and increments this pointer with each new allocation. As a result, Cheney's algorithm works faster than Mark-Sweep by a factor of four for allocation. In all cases tested, average allocation time remained less than 100 nanoseconds for Cheney's algorithm. A second important observation is that the average allocation time is inversely proportional to the size of the heap. This is because a larger heap will require garbage collection pauses to occur less often. This phenomenon holds true regardless of GC strategy.

Garbage Collection Timing:

Cheney's algorithm also significantly outperformed Mark-Sweep for GC pause time, and also proved to be far more stable with respect to heap size. For Mark-Sweep, GC pause time is clearly proportional to heap size, but that effect is less clear for Cheney's algorithm. Cheney's algorithm remains relatively constant regardless of heap size. In all cases tested, the average GC pause time for Cheney's algorithm remained under 30 thousand nanoseconds. Even when the heap size was extended to 1 megabyte in size, and order of magnitude larger than other tested values, the average GC pause time averaged only 50 thousand nanoseconds. In comparison, Mark-Sweep run with a 1 megabyte heap averages 4.9 million nanosecond GC pauses: three orders of magnitude longer than Cheney's algorithm. While not rigorous, this is strong evidence showing that the average performance of Cheney's algorithm is actually sublinear with respect to the heap size. This is largely due to the fact that Cheney's algorithm requires examining only live nodes, while Mark-Sweep must scan through the entire heap and remove dead nodes during the Sweep phase. For our lisp script, the majority of nodes have very short lifetimes, meaning the majority of the heap is filled with dead nodes.

Concluding Statements:

Since a Mark-Sweep collector traverses the heap by following pointers to referenced objects starting at the roots, it is necessary for a Mark-Sweep collector to traverse the heap multiple times in order to mark and sweep objects to be freed. In contrast, a Copy collector copies objects reachable from a root note from the from-space in memory to the to-space as the nodes are traversed with Cheney's bread-first search. As we have stated in this paper, both garbage collectors can be utilized effectively in different environments, or for different languages.

Cheney's sample algorithm :

```
initialize() =
    tospace   = N/2
    fromspace = 0
    allocPtr  = fromspace
    scanPtr   = whatever -- only used during collection
```

```
allocate(n) =
    If allocPtr + n > fromspace+ N/2
        collect()
    EndIf
    If allocPtr + n > fromspace+ N/2
        fail "insufficient memory"
    EndIf
    o = allocPtr
    allocPtr = allocPtr + n
    return o

collect() =
    swap(fromspace, tospace)
    allocPtr = fromspace
    scanPtr  = fromspace

    -- scan every root you've got
    ForEach root in the stack -- or elsewhere
        root = copy(root)
    EndForEach

    -- scan objects in the heap (including objects added by this loop)
    While scanPtr < allocPtr
        ForEach reference r from o (pointed to by scanPtr)
            r = copy(r)
        EndForEach
        scanPtr = scanPtr  + o.size() -- points to the next object in
the heap, if any
    EndWhile


copy(o) =
    If o has no forwarding address
        o' = allocPtr
        allocPtr = allocPtr + size(o)
        copy the contents of o to o'
        forwarding-address(o) = o'
    EndIf
    return forwarding-address(o)
```

PSEUDO - CODE THAT IMPLEMENTS CHENEY'S ALGORITHM

class Object {
// remains null for normal objects
// non-null for forwarded objects

```
  Object* _forwardee;

public:
  void forward_to(address new_addr);
  Object* forwardee();
  bool is_forwarded();
  size_t size();
  Iterator<Object**> object_fields();
};

class Heap {
  Semispace* _from_space;
  Semispace* _to_space;

  void swap_spaces();
  Object* evacuate(Object* obj);

public:
  Heap(address bottom, address end);

  address allocate(size_t size);
  void collect();
  void process_reference(Object** slot);
};

class Semispace {
  address _bottom;
  address _top;
  address _end;

public:
  Semispace(address bottom, address end);

  address bottom() { return _bottom; }
  address top()    { return _top; }
  address end()    { return _end; }

  bool contains(address obj);
  address allocate(size_t size);
  void reset();
};

void Object::forward_to(address new_addr) {
  _forwardee = new_addr;
}
```

```
Object* forwardee() {
 return _forwardee;
}

bool Object::is_forwarded() {
 return _forwardee != nullptr;
}

// Initialize the heap. Assuming contiguous address space
Heap::Heap(address bottom, address end) {
 size_t space_size = (end - bottom) / 2;
 address boundary = bottom + space_size;
 _from_space = new Semispace(bottom, boundary);
 _to_space   = new Semispace(boundary, end);
}

void Heap::swap_spaces() {
 // Swap the two semispaces.

 // std:swap(_from_space, _to_space);
 Semispace* temp = _from_space;
 _from_space = _to_space;
 _to_space = temp;

 // After swapping, the to-space is assumed to be empty.
 // Reset its allocation pointer.
 _to_space->reset();
}

address Heap::allocate(size_t size) {
 return _from_space->allocate();
}

Object* Heap::evacuate(Object* obj) {
 size_t size = obj->size();

 // allocate space in to_space and copy object to there
 address new_addr = _to_space->allocate(size);
 copy(/* to */ new_addr, /* from */ obj, size);

 // set forwarding pointer in old object
 Object* new_obj = (Object*) new_addr;
 obj->forward_to(new_obj);

 return new_obj;
}
```

```
void Heap::collect() {
 // The from-space contains objects, and the to-space is empty now.

 address scanned = _to_space->bottom();

 // scavenge objects directly referenced by the root set
 foreach (Object** slot in ROOTS) {
  process_reference(slot);
 }

 // breadth-first scanning of object graph
 while (scanned < _to_space->top()) {
  Object* parent_obj = (Object*) scanned;
  foreach (Object** slot in parent_obj->obejct_fields()) {
   process_reference(slot);
   // note: _to_space->top() moves if any object is newly copied
   //      into to-space.
  }
  scanned += parent_obj->size();
 }

 // Now all live objects will have been evacuated into the to-space,
 // and we don't need the data in the from-space anymore.

 swap_spaces();
}

void Heap::process_reference(Object** slot) {
 Object* obj = *slot;
 if (obj != nullptr && _from_space->contains(obj)) {
  Object* new_obj = obj->is_forwarded() ? obj->forwardee() // copied
                                        : evacuate(obj);   // not copied (not marked)

  // fixup the slot to point to the new object
  *slot = new_obj;
 }
}

Semispace::Semispace(address bottom, address end) {
 _bottom = bottom;
 _top    = bottom;
 _end    = end;
}

address Semispace::contains(address obj) {
```

```
  return _bottom <= obj && obj < _top;
}

address Semispace::allocate(size_t size) {
  if (_top + size <= end) {
    address obj = _top;
    _top += size;
  } else {
    return nullptr;
  }
}

void Semispace::reset() {
  _top = _bottom;
}
```


SNEHAMOL .J

18BEC054

BSc Electronics and Communication System

Email: snehamolj18bec054@skasc.ac.in