Construcción de un Mark-Sweep Garbage Collector

Comparación Entre Dos Funciones de Marcado

Guido Tagliavini Ponce
Universidad de Buenos Aires
guido.tag@gmail.com

Índice

1. Abstract

Presentamos los conceptos fundamentales involucrados en la construcción de un Mark-Sweep Garbage Collector. Luego de dar un panorama general del tema, presentamos dos funciones de marcado. Ambas ejecutan un DFS sobre el grafo de objetos, pero se diferencian en la cantidad de memoria que utilizan. Mientras que una de ellas no requiere memoria adicional además de la utilizada por la pila de DFS, la otra utiliza, para acelerar el proceso, un conjunto con las direcciones en memoria de los objetos. Calculamos la complejidad temporal de ambos métodos. Presentamos las mediciones de sus tiempos de ejecución realizadas. Terminamos con un análisis de la utilidad práctica de cada versión.

2. Introducción

En ciencias de la computación, un garbage collector (o simplemente GC) es un mecanismo automático para la administración de la memoria. En particular, este mecanismo se encarga de

Figura 1: Grafo de objetos para una lista

liberar los recursos (porciones de memoria) solicitados por un programa, que ya no se encuentran en uso. Esto permite al programador desligarse de la tarea de desalojo de recursos, simplificándole la tarea de programar y depurar. Como contraparte, al tiempo de ejecución de un programa se le suma un tiempo de procesamiento asociado al recolector, haciendo que ésta no sea una técnica apta para aplicaciones cuya performance sea crítica.

El problema fundamental que debe resolver un GC es el de determinar cuáles de los objetos en memoria ya no se encuentran en uso. En un programa cualquiera, se mantienen referencias a objetos que se encuentran en la memoria principal. Durante cierto período de la ejecución esas referencias son útiles, es decir, las utilizamos activamente para acceder a los objetos asociados. Posteriormente, en la misma ejecución, es posible que ya no sea de nuestro interés mantener algunas de esas referencias, y por lo tanto queremos que el recurso apuntado, en caso de que no sea accesible desde otro lugar de nuestro programa, sea desalojado.

Resulta indispensable tener un modelo que provea una representación clara de los objetos en memoria y las referencias existentes entre ellos. Para esto, consideramos un grafo dirigido $G = (V, E)$ cuyos vértices $V = \{o_1, \cdots, o_n\}$ representan los objetos en memoria, y tal que hay una arista $o_i \to o_j$ si y solo si o_i tiene una referencia al objeto o_j. A este digrafo lo llamaremos *grafo de objetos*.

Para ejemplificar, consideremos el caso de una lista simplemente enlazada de 4 nodos. Un grafo asociado a este esquema de objetos es el indicado en la Figura 1. El nodo marcado en rojo tiene una propiedad especial: en general, en una implementación de una lista mantenemos, desde nuestro programa, una referencia directa al primer nodo. En este sentido, decimos que un objeto es *root* si tenemos una referencia directa desde el programa hacia ese objeto. En lo que sigue, siempre distinguiremos a los nodos root con rojo.

Una de las técnicas más conocidas de garbage collection es la llamada *Mark-Sweep*. Específicamente, Mark-Sweep es una familia de métodos, que se componen de dos partes: la primera parte de marcado de objetos muertos, y la segunda parte de barrido o liberación de los recursos asignados a tales objetos. Dependiendo de la implementación de la etapa de marcado y de barrido, obtendremos distintos algoritmos de Mark-Sweep.

En el caso de la Figura 1, si a lo largo del programa se descarta la referencia a o_1 entonces o_1 deja de ser root y todos los objetos pasan a ser inaccesibles y por lo tanto el GC liberará la memoria que ocupan. En el grafo de la Figura 2, al eliminar la referencia a o_1 el objeto o_2 se torna inaccesible y por lo tanto será borrado. Sin embargo todos los demás objetos son accesibles a través de o_4 y deben permanecer en memoria, aún siendo accesibles desde o_2.

Hay situaciones en las que no es tan evidente qué memoria debe liberarse, por ejemplo en el grafo de la Figura 3. Suponiendo que se elimina la referencia al nodo root o_6, ¿qué nodos dejan de ser accesibles? En general, dado un grafo de objetos, al ejecutar el GC los nodos que deben ser barridos son exactamente aquellos que no son alcanzables desde ninguno de los nodos root.

3. Mark-Sweep garbage collection

En esta sección presentamos los rasgos generales del método Mark-Sweep. Para un tratamiento profundo y detallado el lector puede consultar [1].

Como ya hemos dicho, Mark-Sweep se compone de dos etapas. La primera de ellas es el marcado, y consiste en reconocer cuáles objetos son alcanzables desde los nodos root. Para esto, basta utilizar cualquier algoritmo de búsqueda en un grafo (como por ejemplo DFS) partiendo desde los nodos root, marcando cada uno de los nodos visitados, de modo tal que al final del recorrido los nodos no marcados sean exactamente aquellos no alcanzables. La segunda etapa es el barrido y

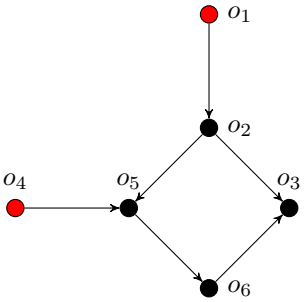

Figura 2: Grafo de objetos con varios nodos root

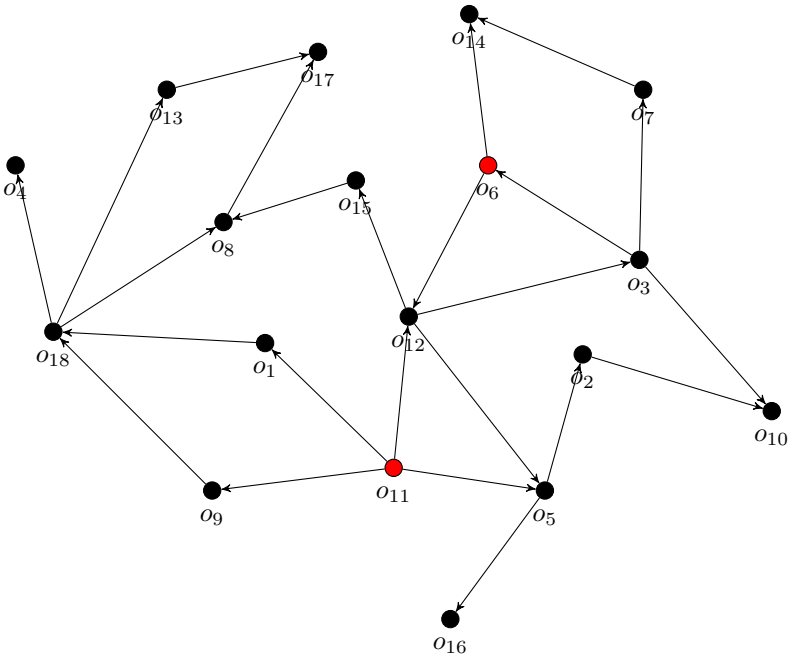

Figura 3: Grafo de objetos complejo

consiste en reconocer los objetos no marcados, y liberar la memoria que ocupan.

3.1. Definiciones

Vamos a definir en forma más precisa algunos términos que hemos venido utilizando en forma intuitiva, para eliminar todo tipo de ambigüedad.

Objeto. Cuando hablamos de un objeto nos referimos a un bloque de datos almacenado en memoria. Consideremos el código,

```
int var;
```

Aquí, el valor almacenado en el espacio asociado a `var` es un objeto.

```
struct data{
    int a;
    char b;
};

struct data *ptr = (struct data *) malloc(...);
```

En este segundo ejemplo, en la dirección almacenada por `ptr` hay un objeto de tipo `struct data`.

Notemos que el primer objeto se almacena en stack mientras que el segundo se encuentra en heap. Como es sabido, la remoción de los objetos en stack ocurre automáticamente al finalizar el bloque en el que se declara, por lo que no hay necesidad de que el garbage collector intervenga en esos casos. Los únicos objetos de los que se ocupa el recolector son aquellos alojados en heap.

Vale la pena destacar que el término *objeto* que utilizamos aquí *no* es equivalente a la noción de objeto dentro de la programación orientada a objetos.

Puntero/Referencia. Si bien existen lenguajes de programación que distinguen estos términos, nosotros los usaremos indistintamente. Cuando estemos hablando de la implementación de una función usaremos el primer término, pues en las implementaciones reales usamos punteros, mientras que cuando hablemos de ideas o conceptos utilizaremos el término referencia.

Grafo de objetos. Dado un conjunto de objetos, definimos el grafo de objetos asociado como un grafo dirigido $G = (V, E)$ con $V = \{o_1, \cdots, o_n\}$ representando el conjunto de objetos y E el conjunto de referencias entre tales objetos. Esto es, $o_i \to o_j \in E$ si y solo si existe una referencia del objeto o_i al objeto o_j. Hablaremos indistintamente de los objetos y sus nodos asociados en un grafo de objetos.

Objeto root. En el contexto de la ejecución de un programa, un objeto es root si se tiene una referencia directa a él desde el programa.

Objecto vivo. Decimos que un objeto está vivo (o que está en uso, o que es accesible) si su nodo asociado en un grafo de objetos es alcanzable desde un nodo root.

Objecto muerto. Decimos que un objeto está muerto si no está vivo.

3.2. Estructuras

Para llevar a cabo las operaciones de marcado y barrido, un GC consta de ciertas estructuras necesarias para mantener información sobre los recursos disponibles y asignados. Con el fin de administrar la memoria, la dividiremos en porciones que llamamos *celdas*. Cada celda se compone de un header más un cuerpo de memoria utilizado para almacenar un objeto. El header contiene tanto información asociada al objeto como campos utilizados por el GC para realizar su cometido. En particular, contiene un campo llamado *mark_bit*, empleado por el GC para determinar si un objeto está o no marcado.

Header	**Objeto**
mark_bit	*str* = "*variable*"
size	*ptr* = 0x1234
...	*var* = 10
	...

Figura 4: Ejemplo de celda

En primer lugar, es necesaria una estructura que mantenga un control de toda la memoria libre de la que se dispone. Para esto se utiliza, típicamente, una lista simplemente enlazada que llamamos *free_list*. Cada nodo de esta lista contiene un puntero a una celda. Cada vez que un usuario solicite memoria, se le asignará una celda de esta lista.

Para mantener registro de las celdas de memoria ocupadas o vivas, se utiliza una segunda lista llamada *live_list*. Cada vez que a un usuario se le es otorgada una celda libre, ésta pasa a la lista de celdas vivas.

Finalmente se requiere de una tercera lista, que indica aquellas celdas que contienen objetos root. Dicha lista recibe el nombre de *root_list*.

Estas estructuras son una necesidad y no una comodidad. Dado que el GC debe recorrer los objetos reservados por el usuario, de no llevar registro de las celdas reservadas debería iterar sobre toda el área de heap y reconocer manualmente los objetos del programa. Esto tiene dos problemas. En primer lugar, el heap podría ser demasiado grande, haciendo muy costosa la etapa de marcado. En segundo lugar, de no contar con ayuda del compilador, el reconocimiento manual de objetos en memoria es difícil y no es completamente efectiva. Un tal reconocimiento se podría llevar a cabo anteponiendo prefijos especiales en las celdas (valores almacenados en memoria al comienzo de una celda) aunque, dado que no existen valores inválidos que no se puedan confundir con esos prefijos, esta técnica no provee ninguna certeza.

3.3. Funciones

Naturalmente, el GC debe proveer una función New que permita reservar memoria, y que devuelva un puntero a la dirección de memoria en la que se encuentra el espacio reservado. Esta función bien podría tomar un argumento que indique el tamaño del bloque de memoria a reservar. En este trabajo hemos decidido ignorar ese argumento, devolviendo siempre celdas del mismo tamaño.

La idea de New es que cuando la memoria libre se haya agotado, i.e. la *free_list* está vacía, se ejecuta el recolector para que devuelva todo el espacio inutilizable a esta lista. Si aún después de correr el GC la lista sigue vacía, entonces no hay objetos muertos y se devuelve Nil. Si no, se devuelve un elemento libre.

La operación New utiliza la función Get-Object que dada una celda devuelve el objeto asociado. El algoritmo devuelve un puntero al objeto y no al principio de la celda donde se encuentra el header, ya que el llamador es ajeno a este encabezado de la celda.

Por otro lado, la lista de nodos root suma un nuevo elemento cada vez que el cuerpo principal del programa adquiere una referencia a un objeto. Dado que nuestra construcción del GC se

input : -
output: o objeto

```
1  begin
2      if free_list = NIL then
3          MARK-SWEEP()
4      end
5      if free_list = NIL then
6          return NIL
7      end
8      Sea c una celda de free_list
9      Borrar c de free_list
10     Agregar c a live_list
11     return GET-OBJECT(c)
12 end
```

Algorithm 1: NEW

input : -
output: -

```
1  begin
2      foreach r ∈ root_list do
3          MARK(r)
4      end
5      SWEEP()
6  end
```

Algorithm 2: MARK-SWEEP

input : c celda de memoria
output: -

```
1  begin
2      if mark_bit[c] = 0 then
3          mark_bit[c] = 1
4          foreach p ∈ pointers(c) do
5              MARK(*p)
6          end
7      end
8  end
```

Algorithm 3: MARK

input : -
output: -

```
1  begin
2      foreach c ∈ live_list do
3          if mark_bit[c] = 1 then
4              mark_bit[c] = 0
5          else
6              Borrar c de live_list
7              Agregar c a free_list
8          end
9      end
10 end
```

Algorithm 4: SWEEP

realiza por fuera de un lenguaje, no podemos realizar esta actualización de nodos root en forma automática. Necesitamos que el programa que utiliza el GC marque explícitamente cuáles objetos serán root.

input : o objeto
output: -

```
1 begin
2     c = GET-CELL(o)
3     Agregar c a root_list
4 end
```

Algorithm 5: SET-ROOT

La operación SET-ROOT utiliza la función GET-CELL que dado un objeto devuelve la celda asociada. Análogamente necesitamos una función para indicar que ya no deseamos mantener una referencia a un objeto.

input : p puntero a un objeto
output: -

```
1 begin
2     c = GET-CELL(*p)
3     Borrar c de root_list
4     p = NIL
5 end
```

Algorithm 6: SET-NULL

La función MARK, encargada de la etapa de marcado, es un DFS sobre el grafo de objetos. Cada vez que visita una celda c, el algoritmo actualiza el *mark_bit* del header e itera sobre el conjunto $pointers(c)$ de punteros que contiene c hacia otras celdas, recorriendo recursivamente cada una de dichas celdas (es decir, nos movemos hacia nodos adyacentes al objeto c en el grafo de objetos). ¿Cómo hace el GC para reconocer estos punteros?

3.4. Detección de punteros

Una característica deseable para un GC es que sea capaz de distinguir punteros en memoria. Esto quiere decir que al analizar el contenido de la memoria en una dirección, sea capaz de determinar si el valor observado es un puntero a un objeto o no. Si el recolector se equivoca, ocurre un error, que puede ser uno de los siguientes:

1. (*Falsos positivos*) **Se determina que el valor es un puntero cuando en realidad no lo es.** Esto implica que como el GC interpreta el valor como una dirección, posiblemente recorra el contenido de la memoria en dicha dirección y la modifique. Esta dirección, aunque no represente un puntero, bien podría ser la dirección de un objeto en memoria.
   - (*Falsos positivos leves*) Si la dirección corresponde al inicio de un objeto, vamos a estar modificando su header. En este caso podríamos estar modificando, por ejemplo, el *marked_bit*, reteniendo memoria que en realidad es ocupada por objetos muertos.
   - (*Falsos positivos graves*) Si la dirección no corresponde al inicio de un objeto, al modificar el contenido de la memoria podríamos estar corrompiéndola y alterando indebidamente la ejecución de un programa.
2. (*Falsos negativos*) **Se determina que el valor no es un puntero cuando en realidad si lo es.** Este error puede ser grave, ya que el objeto apuntado podría terminar no siendo marcado por el GC y, en consecuencia, desalojado.

La capacidad de un GC para detectar punteros en memoria se denomina *type accuracy*. Existen dos tipos de garbage collectors, que se distinguen según esta capacidad. Por un lado están los *type accurate* garbage collectors, que siempre distinguen correctamente los punteros, y por otro lado estan los *conservative* garbage collectors, que son aquellos que no ofrecen garantía de que la detección sea eficaz. En general, los primeros dependen fuertemente de la ayuda que les provea el compilador, mientras que los segundos necesitan poca o ninguna ayuda. Dado que el GC que estamos describiendo es independiente del compilador, la única posibilidad es que sea conservativo.

Para detectar los punteros en una celda, nuestro GC recorre la memoria del objeto asociado, tomando bloques de bytes consecutivos, de tamaño igual al de un puntero en la arquitectura de la máquina (por ejemplo, en AMD64 toma bloques de 8 bytes). Para cada uno de estos bloques, chequea si el valor que contiene es la dirección de algún objeto en la *live_list*. Así aseguramos que el GC solo pueda cometer errores del tipo falsos positivos leves.

```
   input  : c celda de memoria
   output: -
 1 begin
 2     if mark_bit[c] = 0 then
 3         mark_bit[c] = 1
 4         o = GET-OBJECT(c)
 5         p = &o
 6         while p + 8 < &o + size do
 7             if live_list contiene un objeto con dirección p then
 8                 MARK(*p)
 9             end
10             p = p + 1
11         end
12     end
13 end
```

Algorithm 7: MARK

En el Algoritmo 7 presentamos un marcado con la detección de punteros explícita. La constante *size* es el tamaño de un objeto cualquiera, que hemos asumido que es el mismo en todos los casos. El algoritmo asume que los punteros tienen un tamaño de 8 bytes.

3.5. Complejidad temporal

En esta sección estudiaremos la complejidad temporal de las funciones de recolección de basura. Llamemos n a la cantidad de objetos vivos (que es igual a la cantidad de elementos de la *live_list*), y m a la cantidad de referencias entre dichos objetos. En otras palabras, en el grafo de objetos vivos, n es la cantidad de nodos y m la cantidad de aristas.

La complejidad de MARK-SWEEP depende de la función SWEEP y de la etapa de marcado (lineas 2-4 del Algoritmo 2). La función SWEEP es claramente $\mathcal{O}(n)$. Analicemos el costo del marcado. Supongamos que esta etapa se lleva a cabo vía el Algoritmo 7. Calculemos el costo de cada linea de tal algoritmo, entre todas las llamadas que se realizan.

La linea 2 se ejecuta $\mathcal{O}(m)$ veces en total, pues a lo sumo se hace una llamada a MARK por cada arista del grafo de objetos. Las lineas 3, 4 y 5 se ejecutan $\mathcal{O}(n)$ veces y las lineas 6, 7 y 10 se ejecutan $\mathcal{O}(n \cdot size)$ veces. La única linea que consume tiempo no constante es la linea 7; llamemos t a su costo. Entonces la complejidad será $\mathcal{O}(m + n \cdot size \cdot t)$. Una implementación naïve de la linea 7, que recorra toda la *live_list*, consume tiempo $t = \mathcal{O}(n)$, haciendo que la complejidad del marcado sea $\mathcal{O}(m + n^2 \cdot size) = \mathcal{O}(n^2 \cdot size)$.

Para reducir este tiempo tenemos que aminorar el costo $\mathcal{O}(n)$ de cada consulta a la *live_list* en la linea 7. Necesitamos saber, rápidamente, si una dirección dada es la dirección de algún objeto vivo. Con este fin podemos construir un conjunto de direcciones de memoria, con una representación que

permita consultar pertenencia con bajo costo. Consideremos un conjunto representado como trie, que contenga las direcciones de todos los objetos en la *live_list*. Si cada dirección es almacenada en hexadecimal, cada consulta en el trie tiene costo $\mathcal{O}(\ell)$, donde ℓ es la máxima longitud en hexadecimal de una dirección de memoria. Entonces, con esta estructura, el marcado tiene costo $\mathcal{O}(m + n \cdot size \cdot \ell)$, aunque como podemos considerar que ℓ es constante (en una arquitectura 64b, las direcciones tienen una longitud $\ell \leq 16$ caracteres hexadecimales) entonces este nuevo marcado tiene complejidad $\mathcal{O}(m + n \cdot size)$. Observar que, no es razonable suponer que $size$ es constante ya que en ciertas aplicaciones los objetos tienen tamaños del orden de los miles de bytes, o mayores inclusive. El Algoritmo 9 presenta esta nueva versión más veloz. Es importante notar que el costo de construcción del conjunto $\mathcal{O}(n \cdot \ell)$ es absorbido por el total $\mathcal{O}(m + n \cdot size)$.

Si bien el segundo método le gana en costo temporal al primero, su complejidad espacial es mayor, pues utiliza memoria adicional para almacenar el trie.

```
input  : -
output: -
1  begin
2      Sea T un conjunto representado con un trie
3      Insertar en T todas las direcciones de los objetos en live_list
4      foreach r ∈ root_list do
5          MARK(T, r)
6      end
7      SWEEP()
8  end
```

Algorithm 8: MARK-SWEEP

```
input  : T conjunto de direcciones
         c celda de memoria
output: -
1  begin
2      if mark_bit[c] = 0 then
3          mark_bit[c] = 1
4          o = GET-OBJECT(c)
5          p = &o
6          while p + 8 < &o + size do
7              if p ∈ T then
8                  MARK(*p)
9              end
10             p = p + 1
11         end
12     end
13 end
```

Algorithm 9: MARK

3.6. Experimentación

Implementamos en lenguaje C un GC en base a los algoritmos descriptos previamente. Medimos los tiempos de ejecución de las dos versiones de Mark-Sweep dadas por los Algoritmos 2 y 8. Todas las mediciones fueron realizadas en un procesador Intel® Core™ i5-3470 CPU de 3.20GHz, utilizando el registro TSC (Time Stamp Counter) del procesador, que mantiene la cantidad de ciclos de reloj transcurridos desde el último encendido.

Descripción	Algoritmo	Complejidad
Mark-Sweep estándar	2	$\mathcal{O}(n^2 \cdot size)$
Mark-Sweep con conjunto de direcciones	8	$\mathcal{O}(m + n \cdot size)$

Cuadro 1: Versiones de Mark-Sweep implementadas

Se testearon las dos versiones sobre distintos tipos de grafos de objetos. La densidad de todos ellos se determinó a través de un parámetro $p \in [0, 1]$, que representaba la probabilidad de que al tomar (en orden) dos objetos, el primero tuviera una referencia al segundo. Para cada valor de $p \in \{0{,}1; 0{,}25; 0{,}5; 0{,}75; 1\}$, medimos el tiempo de una ejecución de la función de Mark-Sweep, para un grafo de $n \in \{500, 1000, 1500, \cdots, 5000\}$ nodos. En todos los casos, la cantidad de objetos root era aproximadamente $n/100$. Además, en un grafo input de n nodos, el tamaño de un objeto era de $n/100$ campos de datos, siendo cada campo de datos de 8B, i. e. $size = n/100 \cdot 8 = 0{,}08 \cdot n$, es decir que el tamaño de los objetos variaba linealmente con la cantidad de objetos, aunque con una pendiente muy pequeña.

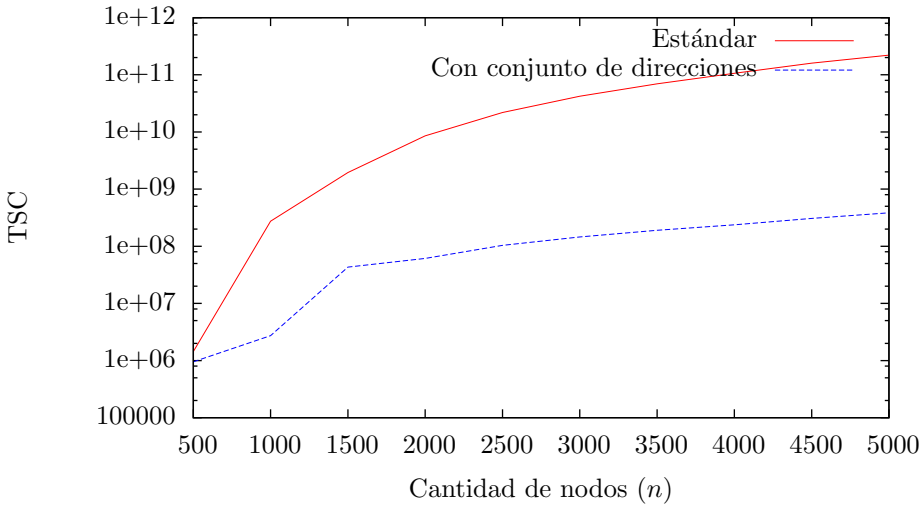

Figura 5: Densidad $p = 0{,}1$

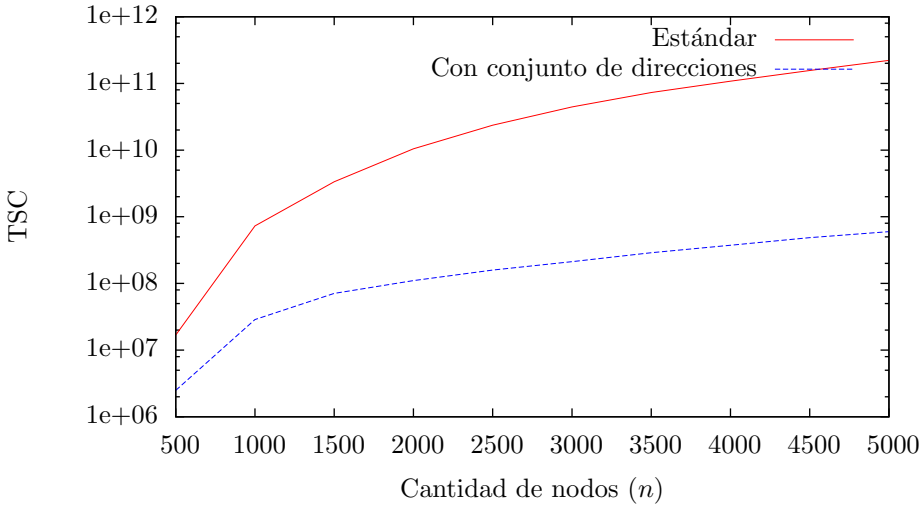

Figura 6: Densidad $p = 0{,}25$

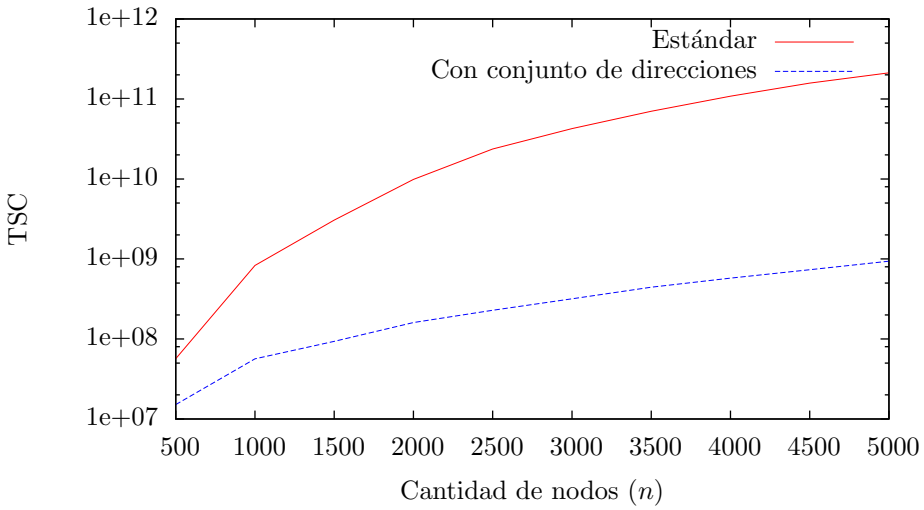

Figura 7: Densidad $p = 0{,}5$

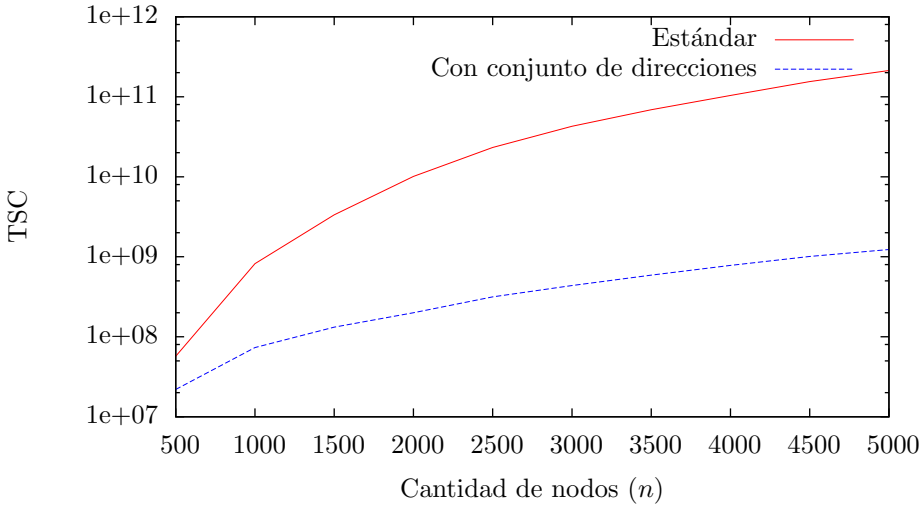

Figura 8: Densidad $p = 0{,}75$

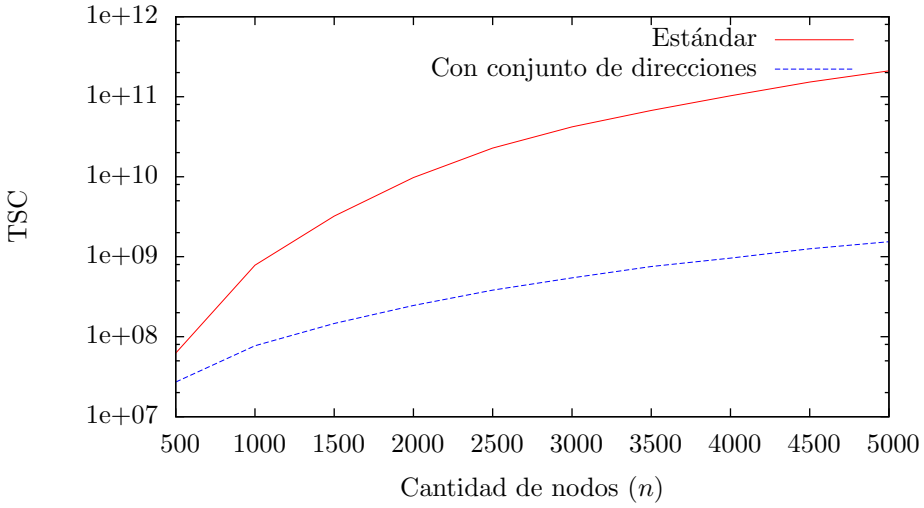

Figura 9: Densidad $p = 1$

Notar que todas las escalas son logarítmicas. La diferencia de tiempos entre ambas versiones es muy grande, siendo el Mark-Sweep con conjunto de direcciones, en promedio, 100 veces más rápida que el estándar.

Se puede observar que la versión estándar no es suceptible ante variaciones en la densidad del grafo de objetos. Esto es coherente con el hecho de que la función es $\Omega(n^2 \cdot size)$ (lo cual se deduce de un análisis de la complejidad análogo al realizado antes) y en consecuencia $\Theta(n^2 \cdot size)$, complejidad que es independiente del valor de m. Contrariamente, la versión de Mark-Sweep con conjunto de direcciones se ve afectada por la densidad (observar la diferencia de crecimento entre las curvas de las mediciones para $p = 0{,}1$ y $p = 0{,}25$). Esto se debe a que tal función es $\Theta(m + n \cdot size)$, dependiente de m.

Medimos además, para la versión con conjunto de direcciones, la memoria que ocupaba el trie representando dicho conjunto. Esta cantidad de memoria depende de varios factores. En primer lugar, depende de cuáles son las direcciones que almacenemos, pues un prefijo común de dos o más direcciones sólo es almacenado una vez en el trie. Claramente, también depende de la cantidad

total de direcciones (igual a la cantidad de objetos). En cierta forma, depende del tamaño *size* de un objeto cualquiera, ya que si los objetos son pequeños y contíguos en memoria entonces sus direcciones en memoria serán similares. Contrariamente, no depende de la densidad del grafo de objetos, pues el conjunto sólo almacena la dirección de cada objeto (de cada nodo del grafo), pero no su vínculo con otros objetos (aristas del grafo). La Figura 10 muestra el requerimiento de memoria promedio para cada tamaño del grafo de objetos.

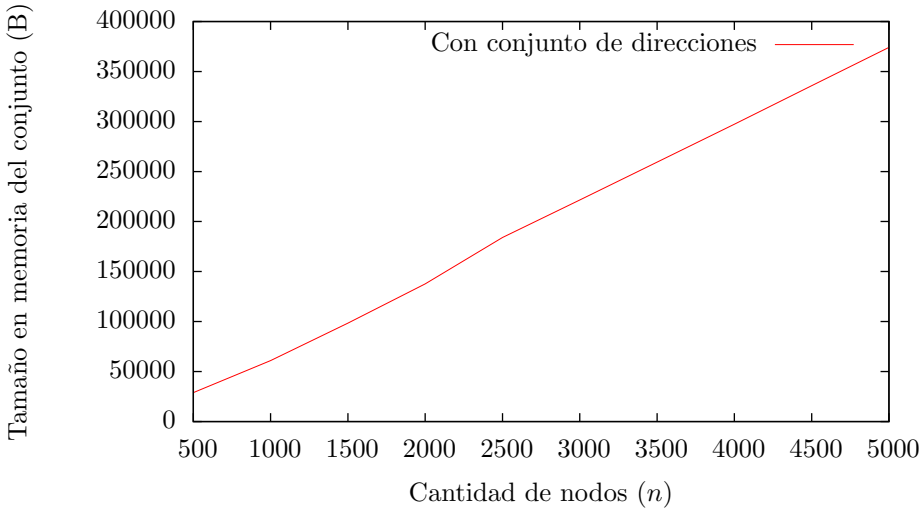

Figura 10: Memoria requerida por la versión con conjunto

Se puede ver que la memoria necesaria crece en forma aproximadamente lineal en la cantidad de objetos. Para 1000 objetos (u 80000B $\approx$ 80KB) , la memoria necesaria es aproximadamente 50KB. Para 2500 objetos (o 500000B $\approx$ 500KB), el requerimiento es de casi 200KB. Para 5000 objetos (o 2000000B $\approx$ 2000KB), el tamaño en memoria termina siendo un poco más de 350KB. Notemos que a medida que la cantidad de objetos crece, el espacio requerido por el conjunto se vuelve despreciable respecto del espacio requerido por los objetos.

De todo este análisis podemos concluir que hay un notable tradeoff entre costo espacial y temporal de una versión a la otra. Mientras que la versión estándar no usa memoria adicional aparte de la empleada por la pila de memoria a lo largo de las llamadas recursivas, es 100 veces más lenta que la versión con conjunto de direcciones, pese a que esta última requiere una notable cantidad de memoria adicional. Esta segunda versión, más rápida pero más costosa en términos espaciales, parece más adecuada en, al menos, dos situaciones:

- Cuando la velocidad de recolección es prioridad por sobre la memoria utilizada.
- Cuando la cantidad de objetos en memoria es muy grande. En este caso, el tamaño del conjunto se vuelve despreciable con lo cual, en términos relativos, la memoria adicional requerida es poca.

En otros casos, la utilización de memoria adicional puede resultar inadmisible, haciendo idónea a la versión estándar.

4. Conclusión

Hemos presentado los componentes básicos de un Mark-Sweep GC. Analizamos dos alternativas para la implementación del marcado, tanto desde el punto de vista teórico como desde el práctico. Concluimos que no hay una implementación superior a la otra para todo escenario posible, si no

que depende de los requerimientos del contexto. La característica de las dos implementaciones es que el talón de Aquiles de una es el punto fuerte de la otra, lo cual lleva a pensar que podría ser interesante la búsqueda de una tercera opción que sea, en algún sentido, intermedia, haciendo un balance entre complejidad espacial y temporal.

Referencias

[1] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.