DISTRIBUTION SWEEPING APLICADO A LA INTERSECCIÓN DE SEGMENTOS ORTOGONALES

Mario Garrido

Universidad de Chile
Departamento de Ciencias de la Computación
Santiago, Chile

ABSTRACT

En el presente trabajo se estudia el desempeño en *I/O* de Distribution Sweeping para resolver el problema de reportar las intersecciones entre segmentos ortogonales, en memoria externa. La cota teórica para *I/O* es $\mathcal{O}(n \log n + \tau)$, lo cual se ve reafirmado por los resultados experimentales, donde es posible apreciar como τ domina cuando el número de intersecciones es muy grande.

Index Terms— *Distribution Sweeping*, *Orthogonal Segment Intersection*, *External Memory*

1. INTRODUCCIÓN

Los objetivos perseguidos en el presente trabajo son los siguientes:

- Realizar una implementación de *Distribution Sweeping* en memoria secundaria, aplicado al problema del reporte de intersecciones entre segmentos ortogonales.
- Comprobar experimentalmente las cotas de *I/O* en memoria secundaria de la implementación y contrastarlas con las cotas teóricas.
- Comprobar experimentalmente los costos de tiempo y su relación con el número de intersecciones encontradas.

1.1. Hipótesis

Las hipótesis a estudiar son las siguientes:

- Cuando el número de intersecciones es bajo, en relación a $n \log n$, entonces el primer término domina $\mathcal{O}(n \log n + \tau)$. En caso contrario es τ quien domina.
- Mientras más bajo sea el valor de α habrá una mayor cantidad de segmentos verticales, lo que se traducirá en menor cantidad de escrituras a disco y menor tiempo.
- Si se modifica el algoritmo para ignorar las recursiones que no contienen segmentos horizontales entonces una distribución de la coordenada x de los segmentos verticales con una media y una varianza dadas generaría una instancia computable más rápidamente que una distribución uniforme.

1.2. Notación

Los siguientes conceptos son de interés tanto para la descripción como para la implementación.

$Memory$: Memoria disponible en el sistema, en *bytes*.
$Filesize$: Tamaño del archivo a procesar, en *bytes*.
$Blocksize$: Tamaño de un bloque, en *bytes*.
$Object\_size$: Tamaño de un objeto. Puede referirse a 4 *bytes*, en caso de que los elementos sean enteros, o a 16 *bytes*, en caso de que los elementos sean líneas.
N: Número de elementos del problema.
T: Número de elementos en la solución del problema.
M: Número de elementos almacenables en $Memory$ *bytes*.
B: Número de elementos almacenables en $Blocksize$ *bytes*.
m: Número de bloques almacenables en $Memory$ *bytes*.
n: Tamaño en bloques del problema.
k: Número de partes de *K-way Mergesort*.
τ: Número de bloques necesarios para almacenar la solución del problema.

2. IMPLEMENTACIÓN

La implementación se desarrolló en el lenguaje *C++*, bajo el standard de *C++* 11. La implementación se encuentra contenida en los archivos *distribution_sweep.cpp*, *distribution_sweep.h*, *experiments.cpp*, *utility.cpp*, *utility.h*, *k_way_mergesort.cpp* y *k_way_mergesort.h*.

2.1. Estructuras de datos y métodos auxiliares

La presente sección describe aspectos relevantes sobre los contenidos de los archivos utility.cpp y utility.h, los cuales corresponden a elementos básicos sobre los que se construyen los algoritmos descritos en las sub-secciones siguientes.

line: La representación de líneas en la implementación es un *struct* que contiene 4 enteros: $\{x_1, y_1, x_2, y_2\}$. Esto permite escribir y leer las líneas a disco fácilmente, ya que cada grupo de 4 números consecutivos representa una línea, si bien no es posible asegurarlo para todos los compiladores de C++, en las pruebas realizadas no se detectó *padding* al escribir a disco. Se incluyen funciones que operan sobre líneas, por ejemplo: *is_horizontal* y *spans_completely*, las cuales nos permiten saber si la línea es horizontal y si ocupa por completo el segmento definido, respectivamente.

array_buffer: Para simplificar el proceso de *I/O*, se implementó la clase *array_buffer*. Esta clase contiene un arreglo (que constituye el *buffer*) de un tamaño dado (el que asumiremos igual a B), un nombre de archivo asociado, punteros a enteros para contar las lecturas/escrituras y otros elementos necesarios para realizar sus funciones. El objetivo de la clase es que las inserciones y extracciones se realicen en bloques de tamaño B, de forma que el costo amortizado de inserción y extracción sea $\mathcal{O}(\frac{1}{B})$, lo que se logra insertando al buffer hasta que se llena, en cuyo caso se vacía su contenido al archivo en disco y se insertan los elementos nuevos desde el principio del buffer (sobreescribiéndolo), y leyendo desde el buffer hasta que se vacía, en cuyo caso se leen B elementos desde el archivo en disco (lo que llena el buffer). Cada vez que se realiza una lectura/escritura se suma uno a los contadores, los cuales son punteros que se le otorgan al *array_buffer* cuando es creado.

Debido a detalles de implementación los *array_buffer* deben trabajar con 2 tipos de datos: líneas (4 enteros) y enteros, por lo que se le debe proveer al *array_buffer* un tipo cuando es creado (0 o 1) y esto determinará las operaciones que utilizará, por esto *write_n*, *read_n*, *retrieve* y *assign* son punteros a funciones. Estos punteros son asignados a las funciones pertinentes, por ejemplo a *write_n* se le asignará *write_n_lines* en caso de estar trabajando con líneas (tipo 1).

min_heap: Para realizar el *k-way merge* en *Mergesort* se implementó la clase *min_heap*. Los *min_heap_node* contienen 2 datos: un puntero al elemento almacenado y un entero que guarda un índice. El *min_heap* contiene un arreglo de nodos, un entero que guarda el tamaño, un tipo y un puntero a función de comparación, esto ya que, al igual que el *array_buffer* debe trabajar con los tipos de datos entero y líneas. Esta estructura nos permite obtener el mínimo con costo de operaciones $\mathcal{O}(1)$ y hacer *min_heapify* con costo de operaciones, en el peor caso, $\mathcal{O}(\log k)$, con k el número de elementos en el *min_heap*.

2.2. External Memory K-way Mergesort

La presente sección describe aspectos relevantes sobre los contenidos de los archivos k_way_mergesort.cpp y k_way_mergesort.h, los cuales corresponden a la implementación del algoritmo K-way Mergesort en memoria externa.

Para ordenar un archivo se llama la función *k_way_mergesort* a la que se le entrega el nombre del archivo, $Memory$, $Blocksize$ y el tipo (líneas o enteros). Esta función calcula $m = \frac{1}{2}\left\lfloor \frac{Memory}{Blocksize} \right\rfloor$ y el número de elementos máximo para cada parte $\left\lfloor \frac{m}{Object\_size} \right\rfloor = \lfloor mB \rfloor$, llamado run_size en el código. Con esto determina el número de *runs* = $k = \left\lceil \frac{N}{run\_size} \right\rceil$. Posteriormente llama a la función *create_runs* y cuando esta retorna llama a *merge_files*.

create_runs: Crea un arreglo de tamaño run_size, crea un *array_buffer* de tamaño B para leer el archivo fuente, y k *array_buffer* de tamaño B, para escribir las piezas.

Posteriormente, lee run_size elementos desde el archivo fuente, a través del *array_buffer*, y se insertan al arreglo, se llama *Mergesort* sobre ese arreglo y se escribe el arreglo ordenado a uno de los *array_buffer*, esto constituye la creación de 1 *run*. Se repite el mismo procedimiento para el resto de los runs. Al terminar se han generado las piezas pre-ordenadas.

mergesort: Divide el arreglo de entrada en 2 partes y se llama recursivamente sobre las mitades, a menos que sea 1 elemento. Posteriormente hace *merge* de las 2 mitades.

merge: Crea una copia de las 2 mitades y las escribe al arreglo original de forma ordenada. Para esto va comparando los elementos de ambas mitades y escribiendo el menor. La creación de la copia es lo que motiva el término $\frac{1}{2}$ en el cómputo de m.

merge_files: Crea k *array_buffer*, uno por cada parte, crea un *array_buffer* para escribir el archivo final y crea un *min_heap* de tamaño k, que contiene un nodo con el primer elemento de cada pieza. Los nodos contienen, además, un *index* que permite identificar la pieza que representan. Para construir el archivo final se selecciona el menor nodo del *heap*, con costo $\mathcal{O}(1)$ de operaciones, y se agrega su elemento al *array_buffer* de salida. Posteriormente, se reemplaza el elemento del nodo con el siguiente elemento del archivo correspondiente y se realiza *heapify*, con costo de operaciones, a lo más, $\mathcal{O}(\log k)$. Esto se realiza para cada uno de los N elementos. Si bien, el costo de operaciones es, a lo más, $\mathcal{O}(N \log k)$, el costo de *I/O* es, a lo más, $\mathcal{O}(n)$, ya que leemos los N elementos en bloques de tamaño B y los escribimos en bloques de tamaño B.

2.3. External Memory Distribution Sweep

La presente sección describe aspectos relevantes sobre los contenidos de los archivos distribution_sweep.cpp y dis-

tribution_sweep.h, los cuales corresponden a la implementación del algoritmo Distribution Sweep, para la intersección de segmentos ortogonales, en memoria externa.

Para comenzar el proceso se llama a la función *orthogonal_segment_intersection* y se le entrega $Memory$, $Blocksize$ y el *sourcefile*. Esta función calcula M, B_{ints} y B_{lines}, que corresponden al número de enteros y de líneas almacenables en un bloque, respectivamente. Procede a crear los *sweep files*, correspondientes a X e Y, llamando a la función *make_sweep_files*. Posteriormente comienza la recursión llamando a *recursion_step*, entregándole B_{ints}, B_{lines}, M, X, Y y un par de índices que codifican los elementos en X a considerar para confeccionar los *slabs*, que en el caso de la primera recursión corresponden a 0 y $N_x - 1$, con N_x := número de elementos en X.

make_sweep_files: Genera el archivo X que contiene todas las coordenadas x de las líneas, ordenadas de forma creciente utilizando *Mergesort*. Posteriormente agrega un elemento a X correspondiente a $max(x) + 1$. Finalmente, ordena el *sourcefile* utilizando *Mergesort*, respecto a la coordenada y_1. Debido a que el archivo Y es un archivo de líneas y X es un archivo de enteros es que *Mergesort* y los *array_buffer* deben ser capaces de trabajar tanto con enteros como con líneas.

recursion_step: Evalúa si $2|Y| \leq M - B_{lines}$, en cuyo caso llama a la función *main_memory_step* para ejecutar el procedimiento en memoria principal. En caso contrario, intenta generar *slabs* en base al archivo X, entre los índices (i, j) dados, llamando a *choose_xi*. Las estrategias utilizadas para elegir la cantidad de *slabs* a generar son descritas en la sección experimental. Por cada *slab* efectivo se crea un *array_buffer* de tamaño B_{lines} que representa la *active list* del *slab*, un *array_buffer* de tamaño B_{lines} que representa la *active list* de reemplazo o nueva *active list* y un *array_buffer* de tamaño B_{lines} que representa Y_i := el conjunto de líneas ordenadas por su coordenada y_1 correspondientes al *slab*. Se crea un último *array_buffer* de tamaño B_{lines} para leer Y.

Para cada línea en Y se determinan los *slabs* a los que pertenecen sus *endpoints*. Si la línea es vertical se agrega a la *active list* del *slab* correspondiente y al Y_i del *slab*. Si es una horizontal se reportan todas las intersecciones con las líneas presentes en las *active list* de los *slabs* cubiertos completamente por la línea, descartando aquellos que ya no deberían estar activos, ya que su coordenada $y_2 < y_{horizontal}$ de la horizontal. Mientras se va recorriendo cada *active list*, todos los elementos que siguen activos se escriben a la nueva *active list* y, una vez que ha terminado el proceso, se borra la *active list* original y se cambian los punteros, con esto se logra la actualización *lazy* de las líneas activas, sin necesidad de considerar como eventos de la *sweep line* las coordenadas y_2 de las líneas. La línea horizontal se agrega a los Y_i de todos los *slabs* que no son cubiertos completamente por ella.

Finalmente, se llama *recursion_step* por cada *slab*. Se le pasa M, B_{ints}, B_{lines}, X, Y_i, que será Y de esa recursión, correspondiente a las líneas ordenadas por y_1 contenidas dentro de los límites del *slab* y se le pasan los índices (i, j) entre los que escoger las cotas de los *slabs* de la recursión, en el archivo X.

main_memory_step: Crea un *array_buffer* de tamaño B_{lines} para leer Y, uno de tamaño N_y para la *active list* y uno de tamaño N_y para la *active list* de reemplazo o nueva *active list*, con N_y := número de líneas en Y.

Se leen las líneas de Y y, para cada una, si es vertical se agrega a la *active list*, en caso contrario se reportan las intersecciones con todos los elementos de la *active list* que siguen activos (cuya coordenada $y_2 \geq y_{horizontal}$) y se genera la nueva *active list* con las líneas todavía activas. Vale la pena notar que, en el caso promedio, una *active list* no contendrá N_y líneas, pero estos tamaños son necesarios para evitar problemas de memoria en el peor caso (todas las líneas son verticales). Una posible estrategia, para permitir que la cota de procesamiento en memoria interna ($2|Y| \leq M$) sea más laxa, es crear un *buffer* de tamaño $\frac{N_y}{2}$ y aumentarlo en caso de que sea necesario, ya que la cota viene por el tamaño de los 2 *array_buffer* necesarios para albergar la *active list* y la *active list* de reemplazo.

choose_xi: Calcula un número tentativo de elementos por *slab*, *elements_per_slab*, y escribe el primer entero (i) como la cota inicial del primer *slab*. Estas cotas se escriben en un arreglo arr, en que $slab_i$:= $[arr[i], arr[i + 1])$. Lee todos los enteros entre en los índices $\in [i, j + 1]$ y por cada *elements_per_slab* enteros leídos intenta escribir el entero como la siguiente cota (el siguiente elemento de arr, que corresponde a la cota exclusiva derecha de un *slab* y la cota inclusiva izquierda del siguiente *slab*), a menos que sea parte de una secuencia de números iguales hacia atrás, en cuyo caso escoge el siguiente elemento distinto. Si, al terminar la lectura, no se han logrado escoger suficientes cotas para definir todos los *slabs*, entonces se agrega como última cota el entero j+1 y se retorna el número de *slabs* efectivos construidos. De aquí que el único elemento que nunca pertenecerá al interior de un *slab* será el último elemento del archivo X, que fue agregado artificialmente y no representa a una línea.

Este procedimiento permite manejar valores duplicados en la entrada y constituye una aproximación a una división de los *slabs* con misma cantidad de líneas, ya que cada vertical es representada por una coordenada x pero una horizontal es representada por 2 coordenadas x, esto abre la posibilidad de que ambas coordenadas x, pertenecientes a una misma horizontal, pertenezcan al interior del mismo *slab* y que, por lo tanto, el número estimado de líneas en ese *slab* sea menor

que el esperado. Vale la pena notar que una división en que el archivo X se compone de líneas ordenadas por x_1 o por x_2 también sufre un problema de mala representación, ya que se ignora una coordenada del todo. En la presente implementación se logra, al menos, considerar el espacio de $\hat{x}$ de forma completa, sin ambigüedades.

3. RESULTADOS EXPERIMENTALES

Los experimentos fueron realizados en *Ubuntu* 14.04, con procesador AMD Phenom II X4 955 corriendo a 3.2 GHz, 26624MB de RAM DDR3 a 1333 MHz, con un SSD Crucial BX100.

Para efectuar los experimentos se utilizó la rutina contenida en el archivo *experiments.cpp*, en el cual es posible modificar todos los parámetros necesarios. En primera instancia era necesario decidir un *Blocksize*, para esto se consultó la información tanto del sistema operativo como del fabricante, los que indicaron 512 *Bytes* y 4096 *Bytes*, respectivamente. En las figuras 1 a 3 es posible apreciar como el cambio del *Blocksize* afecta el tiempo. Si bien, aumentar el *Blocksize* siempre produjo un menor tiempo (se experimentó hasta 16384 *Bytes*), se optó por el tamaño sugerido por el fabricante ya que se buscaba eliminar optimizaciones realizadas por el sistema operativo (aprovechando que los *array_buffer* escriben directamente a disco, por un lado, y por otro lado, era problemático hacer que los experimentos arrojaran datos significativos si considerar una memoria de 1 bloque era más grande que la instancia del problema a resolver (para tamaños bajos, como 2^9).

Una situación que se encontró en la experimentación, es la que se ilustra en las figuras 4 y 5: El sistema operativo tiene un *overhead* considerable cuando debe escribir y leer a *NTFS*. Originalmente las pruebas se realizaron en una partición *NTFS*, contenida dentro del mismo disco, pero estas pruebas eran, en general, cerca de 4 veces más lentas que en *EXT3/EXT4*, por lo que fue necesario reiniciar los experimentos.

Contando con un *Blocksize* de 4096 *Bytes* se prosiguió a utilizar una estrategia de *Branching* (número tentativo de *slabs*) fijo: En cada recursión se intenta crear 4 *slabs* para dividir el problema. Originalmente, la motivación de esto era evitar una división excesiva del problema, lo que produciría un costo innecesario de *I/O*. En las figuras 6 a 18 se aprecia la información relevante obtenida de los experimentos. En las figuras 11 y 12, se aprecia como la curva *linearítmica* puede acotar superiormente el número de *I/O* de *Distribution Sweep* (excluyendo *I/O* de *Mergesort* y del reporte de intersecciones). Como las intersecciones se comportan, también, como una curva linearítmica, se tiene que $\frac{T}{B}$ es también linearítmica y, de acuerdo, a la información de las figuras 14 a 18, notamos que la cantidad de *I/O* siempre termina siendo dominada por *Distribution Sweep*.

En general, la distribución binomial produce mayor número de intersecciones, debido a la concentración de los valores generados en torno a una media. Para aproximar el comportamiento de una distribución normal se utilizó una distribución binomial con parámetros INT_MAX (valor máximo) y p = 0.5, lo que emula una distribución normal con media $\frac{INT\_MAX}{2}$ y varianza, aproximadamente, 500000000. El mayor número de intersecciones, junto con una mayor densidad de los datos en torno a la media, hace que la cantidad de *I/O* sea también mayor en el caso de la distribución binomial. En ambas distribuciones se tiene a que cerca del 80 % del *I/O* sea de *Distribution Sweep* y cerca de un 20 % sea por el reporte de intersecciones (las intersecciones se escriben a disco, para que $\frac{T}{B}$ tenga sentido). La figura 10 muestra, como ya era posible inferir, que el tiempo de *Mergesort* es negligible conforme el tamaño de la muestra aumenta. La memoria asignada a cada experimento corresponde a una décima parte de la memoria necesaria para almacenar las muestras.

Posteriormente se consideró otra estrategia de *Branching*, en que se genera la mayor cantidad de *slabs* posibles, considerando que se necesitan 3 bloques por cada slab (para los buffers) y 2 bloques para los buffers de lectura de la *Sweep line* y de escritura de las intersecciones. Para que los experimentos tuvieran sentido, se le asignó 32768 *Bytes* de memoria a los tamaños ≤ 12 (lo que les permite tener 2 *slabs* por recursión), y 57344 *Bytes* para los tamaños ≤ 15, lo que les permite tener 4 *slabs*. De ahí en adelante se asigna, como en la estrategia anterior, un décimo de la memoria necesaria para albergar los datos del problema. Esta estrategia hace uso de un *Branching* agresivo. Si bien, no fue posible experimentar esta estrategia a fondo, se aprecia en las figuras 19 a 21 una comparación de los datos disponibles, con respecto a la estrategia anterior, con la salvedad de que para el tiempo se descarga el tiempo de reporte de las intersecciones (solo corresponde a *Distribution Sweep*, a diferencia de las figuras 8 y 9). Se aprecia que, para el caso de la binomial, la nueva estrategia resulta prometedora, respecto a los resultados del tamaño 19 en la figura 21. Pese a que el *I/O* es menor en la uniforme (figura 20), los tiempos no necesariamente son mejores (figura 19), aunque si es necesario conjeturar, pareciera que a mayor tamaño de los datos y mayor tamaño del *Branching* se van obteniendo diferencias más marcadas, para los casos de α = 0.25 y 0.5, como se evidencia en la figura 19, particulamente en el único dato del *Branching dinámico* de tamaño 20.

Se adjuntan todas las figuras en una carpeta, para su mejor análisis.

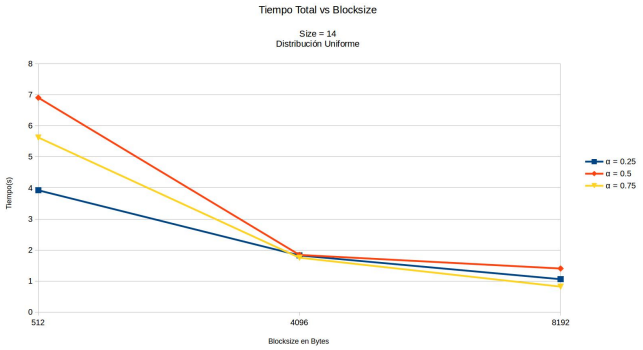

Fig. 1. Tiempo vs Blocksize - Distribución Uniforme - Size 14

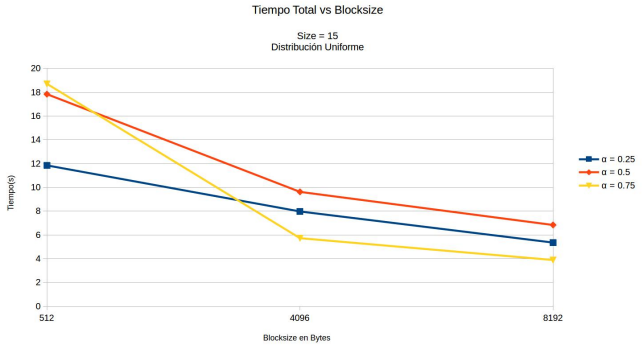

Fig. 2. Tiempo vs Blocksize - Distribución Uniforme - Size 15

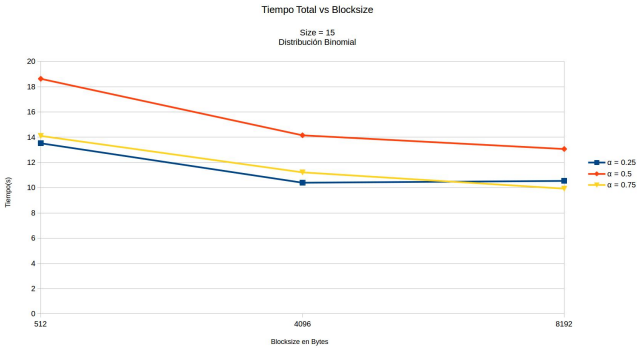

Fig. 3. Tiempo vs Blocksize - Distribución Binomial - Size 15

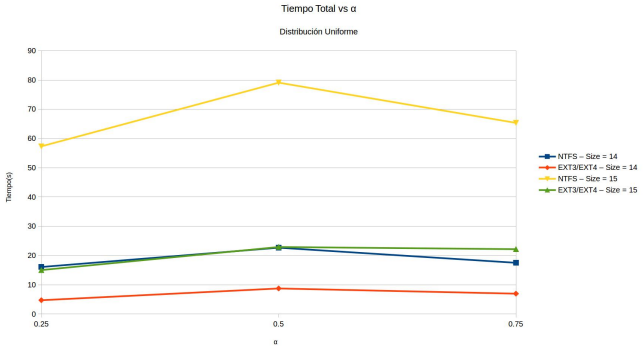

Fig. 4. Tiempo vs α - Distribución Uniforme - Formato

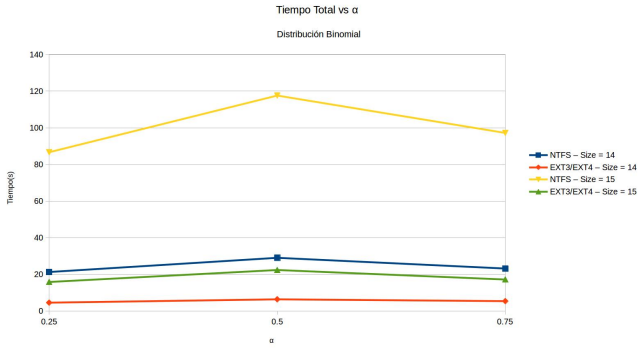

Fig. 5. Tiempo vs α - Distribución Binomial - Formato

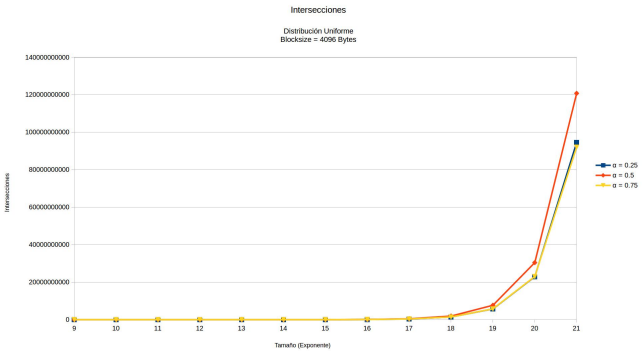

Fig. 6. Intersecciones vs Tamaño - Distribución Uniforme

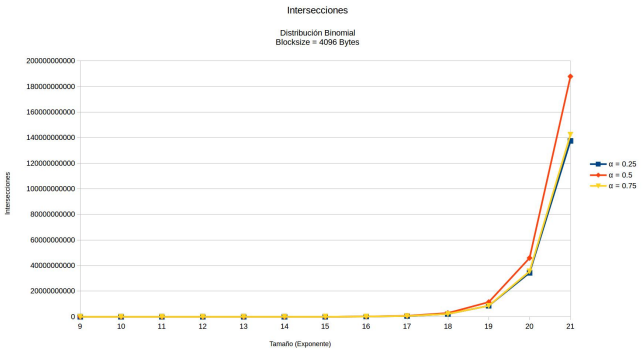

Fig. 7. Intersecciones vs Tamaño - Distribución Binomial

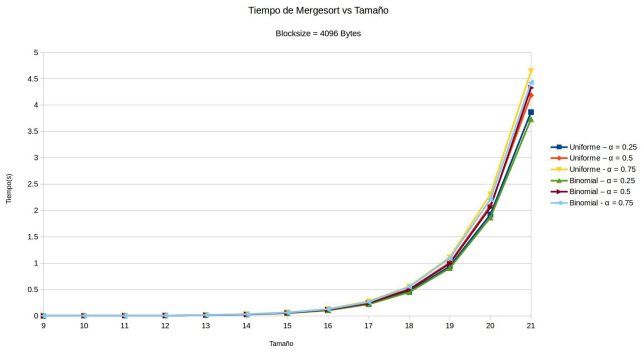

Fig. 10. Tiempo de Mergesort vs Tamaño

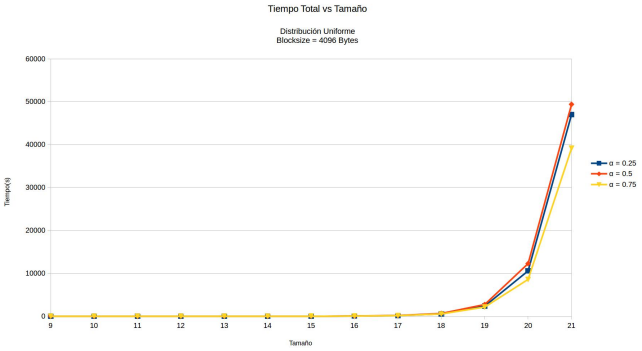

Fig. 8. Tiempo total vs Tamaño - Distribución Uniforme

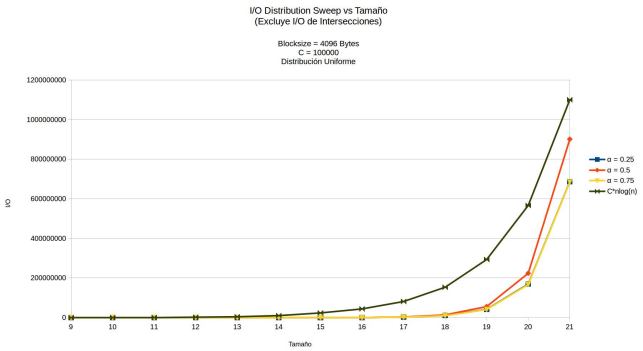

Fig. 11. *I/O* Distribution Sweep vs Tamaño - Distribución Uniforme

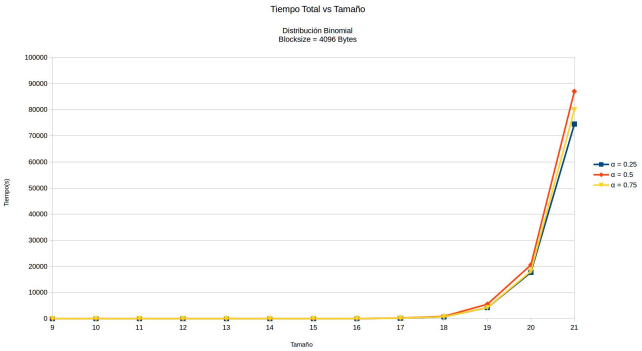

Fig. 9. Tiempo total vs Tamaño - Distribución Binomial

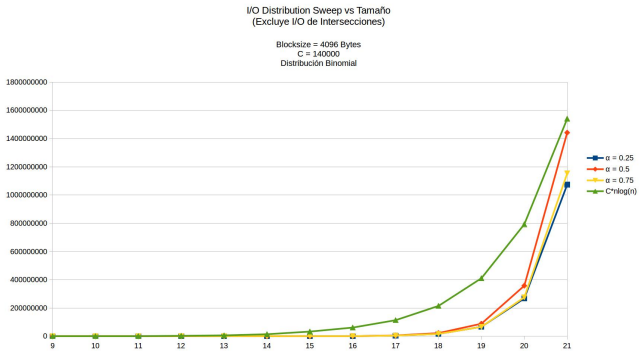

Fig. 12. *I/O* Distribution Sweep vs Tamaño - Distribución Binomial

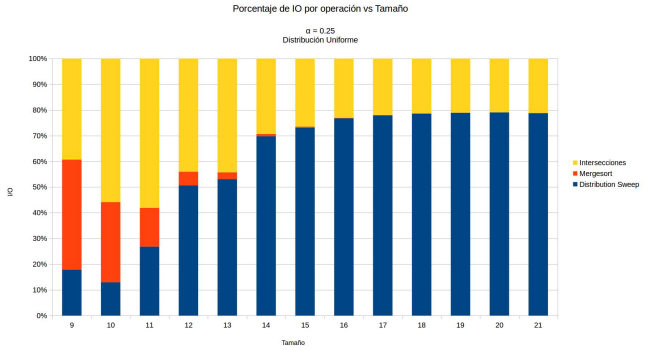

Fig. 13. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.25$, Uniforme

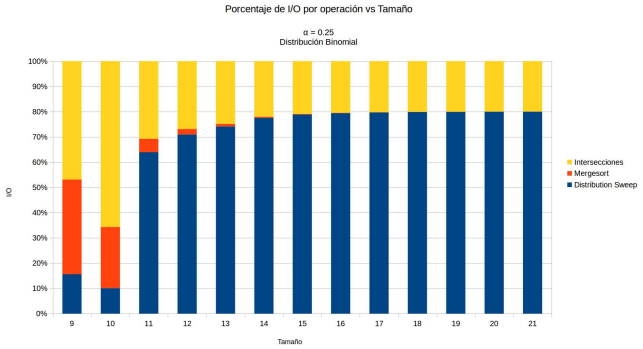

Fig. 16. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.25$, Binomial

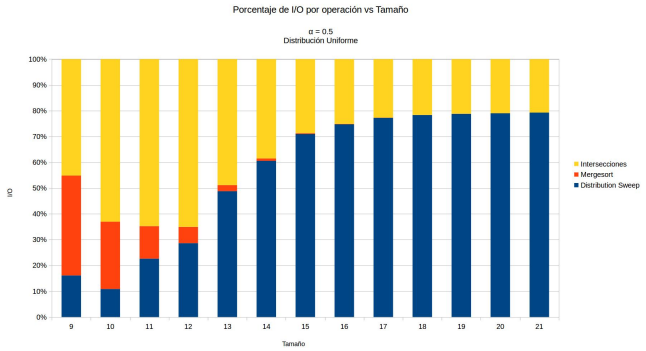

Fig. 14. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.5$, Uniforme

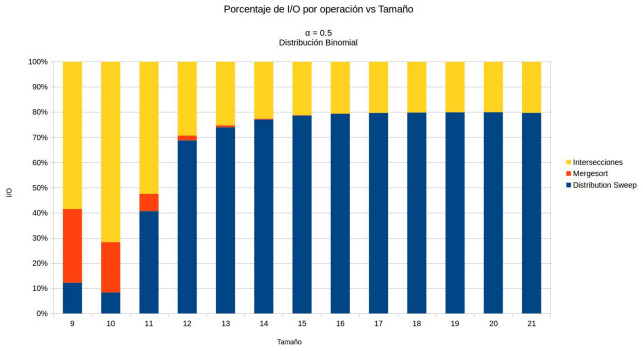

Fig. 17. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.5$, Binomial

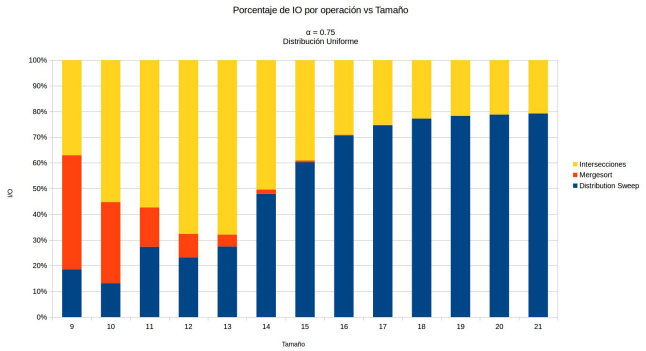

Fig. 15. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.75$, Uniforme

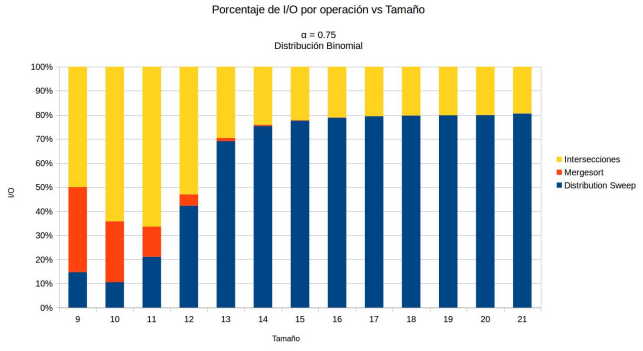

Fig. 18. Porcentaje de *I/O* vs Tamaño - $\alpha = 0.75$, Binomial

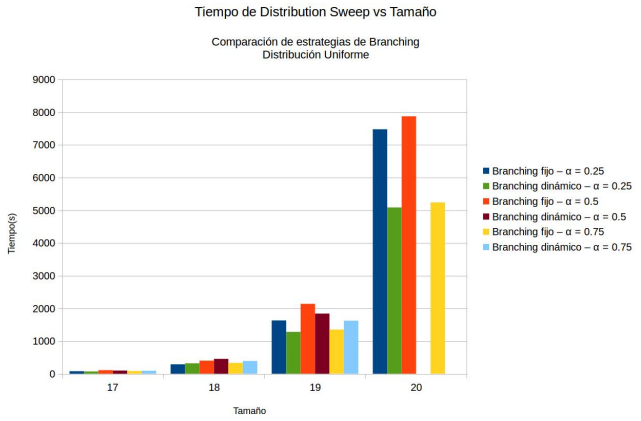

Fig. 19. Tiempo de Distribution Sweep vs Tamaño - Comparación de estrategias de Branching - Uniforme

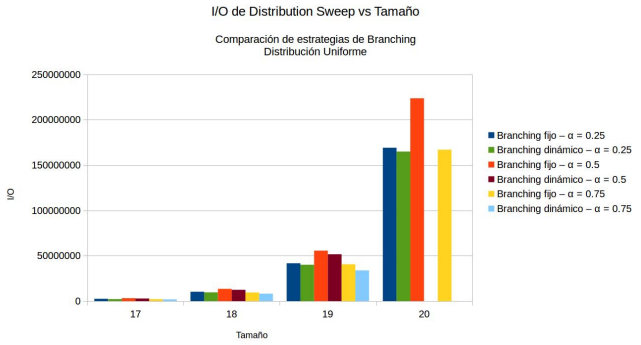

Fig. 20. *I/O* de Distribution Sweep vs Tamaño - Comparación de estrategias de Branching - Uniforme

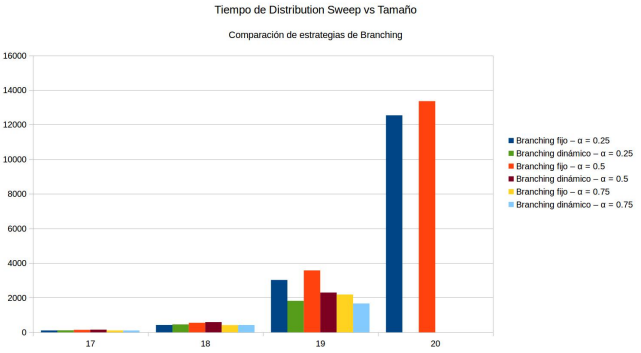

Fig. 21. Tiempo de Distribution Sweep vs Tamaño - Comparación de estrategias de Branching - Binomial

4. CONCLUSIONES

La función linearítmica domina frente al reporte de intersecciones, ya que el tamaño del bloque permite escribir muchos más enteros que líneas y, además, dada la generación de los datos se tiene que las intersecciones se comportan también de forma linearítmica. El valor de α más difícil de manejar para el algoritmo fue 0.5, y contrario a lo que se creía tanto 0.25 como 0.75 producen resultados similares. No se exploró la opción de ignorar los *slabs* que no contienen segmentos horizontales.