Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC4102 Diseño y Análisis de Algoritmos
Prof: Gonzalo Navarro B.

Diccionarios en Memoria Secundaria
Tarea 2
CC4102 - Diseño y Análisis de Algoritmos

Nicolás Salas V.
Daniel Rojas C.

23 de noviembre de 2015

Índice general

1. Introducción

El objetivo de esta tarea -y por tanto de este informe- es estudiar estructuras de datos en memoria secundaria.

En el contexto de esta tarea, con memoria secundaria se quiere decir esencialmente *disco*[1]. Esto es, sabiendo que algunas estructuras de datos son mejores para disco que otras, lo que interesa es verificar:

1. Si efectivamente son buenas estructuras para disco.

2. Dar una medida del espacio de las estructuras ocupan en disco.

Antes de continuar, es bueno dejar en claro algunas cosas y supuestos que pueden parecer obvios, pero siempre es mejor aclarar. Muchas de las cosas que en este informe se explican se toman por sentadas muchas veces sin dar prueba de ello. Dichas pruebas escapan al alcance de este informe, pero el hecho de declarar algunos resultados sin sus correspondientes pruebas no hace que sean menos verdaderos. De nuevo, dichas demostraciones **escapan** al alcance de este informe, sobre todo en esta sección introductoria.

No obstante, para los objetivos, resultados e interpretaciones que sí incumben a este informe, por cierto que se presentarán pruebas experimentales y teóricas cuando sea necesario.

En Memoria Secundaria, siempre se asume que la cantidad de datos es gigantesca y por tanto mucho, mucho mayor que la cantidad de datos que cabe en memoria principal, de manera que los algoritmos y estructuras de datos convencionales usados para resolver en memoria secundaria pueden o no ser los mejores. Un buen ejemplo de esto es *MergeSort*, que no es el mejor algoritmo para ordenamiento en memoria principal, pero ciertamente que es de los mejores en memoria secundaria.[2]

Entonces, se sabe[3] que las operaciones en disco son *bastante* más lentas en memoria secundaria que en memoria principal. Por lo tanto, para los objetivos de este informe no se considerará ningún procesamiento en memoria principal, la razón de esto es precisamente lo anterior: como el una lectura o escritura de memoria secundaria es bastante más lenta que una lectura o escritura de memoria principal, el tiempo asociado al procesamiento en memoria principal es nulo comparado con el tiempo que toma cualquier operación en memoria secundaria. Esto es lo que se conoce como **I/O Model**.

El enfoque de este experimento consiste en medir las operaciones en disco y la efectividad de las estructura de datos que se van a probar según I/O Model.

---

[1] Es un poco intrincado decir que memoria secundaria *es* disco. Pero digamos que una operación de memoria secundaria es una operación en disco. Con operación hablamos de Lectura o Escritura.

[2] Lo dejamos sin pruebas, pero una pequeña búsqueda en `https://scholar.google.cl/scholar?q=mergesort` puede llevar a pruebas fehacientes de que lo que decimos es cierto.

[3] De nuevo, esto es cierto, pero lo declaramos sin pruebas.

1.1. Estructuras de Datos

Hasta ahora se ha dado una discusión bastante grande de qué es memoria secundaria, cómo opera y el modelo en el que se va a circunscribir el experimento. Las estructuras de datos que se van a comparar en este informe son:

- Árboles B
- Hashing Extendible
- Hashing Lineal

Árbol B

A grandes rasgos, un Árbol B es un árbol $B+1$-*ario* donde cada nodo tiene B elementos[4]. La condición que cumple un Árbol B es que el k-ésimo elemento de un nodo es mayor que todos los elementos del k-ésimo hijo y menor que todos los del $(k+1)$-ésimo. Adicionalmente, en memoria secundaria, y para asegurar que el árbol no gasta demasiado espacio, se impone que todos los hijos tengan $\lfloor \frac{B}{2} \rfloor$ elementos. Mayores detalles respecto de estas estructuras se dan en la sección de Diseño Experimental.

Hashing Extendible

Una estructura de hashing extendible es una estructura que, como cualquier otra de hashing, está diseñada para hacer búsquedas en tiempo $O(1)$ usando una función de hashing. Para ello se usan los bits generados por la función de hashing y un *Trie*[5] para navegar por los elementos insertados **en memoria principal**. Las hojas del trie apuntan a páginas del disco que contienen los elementos. Recuérdese que cuando se habla de memoria secundaria, $O(1)$ corresponde a la cantidad de **accesos a disco**, y no a la cantidad de comparaciones. Mayores detalles se dan en la sección de Diseño Experimental.

Hashing Lineal

El hashing lineal, al igual que el Hashing Extendible, intenta realizar la mínima cantidad de accesos a disco posible. Para divide los elementos según el módulo de su función de hashing en algún número S y los separa según el resultado de dicha operación. Esta estructura tiene la ventaja (sobre el hashing extendible) de no necesitar de memoria principal para mantener su consistencia en disco. Esta estructura es más complicada de explicar simplificadamente, de manera que se dejará su explicación a fondo para la sección de Diseño Experimental.

1.2. Notación

Frecuentemente en este informe se hará referencias a las cantidades (o símbolos) B, M y N. Para aclararlo se ha hecho una pequeña tabla que clarifica esto.

[4] Algunos autores usan B como otro parámetro (por ejemplo, la profunidad del árbol). Pero la estructura es la misma de todas maneras

[5] En los apéndices se describe esta estructura.

Símbolo	Significado
B	Tamaño del Bloque en disco
M	Tamaño total de la Memoria Principal
N	Tamaño del input de un problema

De no especificarlo, estas cantidades siempre estarán en bytes. Adicionalmente, cualquier otro símbolo que se introduzca se especificará según corresponda.

1.3. Experimento

Para ver qué estructuras son mejores según las medidas de rendimiento antes expuestas se parametrizará el tiempo según la cantidad de accesos a disco[6].

Interesa, entonces, comparar el uso efectivo de memoria secundaria versus el uso real de disco y verificar si es que existe alguna relación entre esas dos cantidades. Los detalles de cómo se hará esta comparación se exponen en la sección de Diseño Experimental.

[6]Técnica bastante común en el análisis de algoritmos.

2. Hipótesis

Una duda bastante legítima, en primer lugar, es preguntarse **por qué** alguien usaría un Árbol B en vez de cualquier estructura de Hashing. En los apéndices se explica que el costo de accesos a disco en un Árbol B es $O(\log_B N)$ amortizado. Mientras que ya se dijo[7] que el costo de accesos en una estructura de Hashing es $O(1)$.

No es difícil darse cuenta que ambas estructuras de Hashing que se prueban para este experimento generan una partición del estilo de *clases de equivalencia* dentro de los posibles inputs, y mientras más se expanden la estructuras, más específica se hace la partición. Esto ciertamente que minimiza la cantidad de accesos a disco por búsqueda, borrado e inserción, pero también hace que se almacenen menos elementos por página mientras más elementos se insertan. Por otro lado, el Árbol B no genera ningún tipo de partición sobre el conjunto, pues todos sus nodos almacenan elementos, así que el Árbol B *no se degenera* cuando la cantidad de elementos se aproxima al máximo número de elementos posible.

Lo anterior permite elaborar la primera hipótesis sobre el experimento.

Existe un trade off entre búsqueda y eficiencia de almacenamiento.

Además, sin lugar a dudas que se tratará de probar experimentalmente que **las búsquedas son más lentas en árboles B que en ambas estructura de hashing**.

Respecto de las diferencias entre ambas estructuras de hashing, el extendible debería ser más rápido que el hashing lineal, porque todo el procesamiento de lo que se denomina un *bucket* es hecho en memoria principal, mientras que en el hashing lineal, el procesamiento de los buckets es hecho en memoria secundaria, lo que implica mayor cantidad de accesos a disco en esta estructura.

[7] Sin pruebas, pero la dejaremos en los apéndices

3. Diseño Experimental

3.1. Lenguaje de Preferencia

Para implementar las estructuras y hacer los experimentos se ha escogido el lenguaje `C`. Esto porque, en general, los experimentos en memoria secundaria tienden a ser muy largos y un lenguaje como C permite minimizar el tiempo de ejecución, de manera que se pueden obtener los resultados rápidamente, al contrario de lo que pasaría con `Java`. Se puede también dar una discusión de por qué `C` y no `C++`, en este caso simplemente se usa C porque no se necesita la orientación a objetos para resolver un problema de memoria secundaria.

3.2. Limitaciones de la Implementación

El trabajo de experimentar con memoria secundaria es, en general, bastante más complejo que lo que en este experimento particular se hace. Hacer un experimento formal para memoria secundaria involucra aspectos de Sistemas Operativos que escapan al alcance de un curso de Diseño y Análisis de Algoritmos de pregrado. En particular, se requeriría un driver para algún sistema operativo que pudiera escribir un bloque *real* de disco y eliminar una cantidad arbitraria de bytes de un archivo.

Considerando las limitaciones que ofrecen las llamadas a sistema de C, el esquema será el siguiente:

- Una estructura será un montón de archivos que almacenan la información de cada página de disco
- Escribir y modificar un bloque se hará vía las llamadas `fwrite` y `fseek` (que internamente usan la llamada a sistema `write` y algunas otras más).

Aunque el análisis parezca pesimista, realmente estas limitaciones no son de tanta urgencia como parece. En este trabajo se medirá numero de lecturas y escrituras desde/hacia disco y la cantidad de espacio que ocupa cada estructura, de manera que no es tan relevante si se escriben o no bloques reales de disco.[8]

Asimismo, el espacio no tiene que ver con la cantidad de los bloques, sino que con su contenido, y esto es específico de la implementación como también independiente de dónde esté la información.

3.3. Estructuras en Memoria Secundaria

Para entender mejor los resultados que se presentan en la siguiente sección, en esta parte se describen todos los detalles de implementación de las estructuras de datos que se usan en los experimentos.

Es natural que los resultados puedan variar mucho según la forma en la que se implementan las estructuras de datos en memoria secundaria, y es justamente por eso que leer esta

[8] Y es de esperar, en todo caso, que el sistema operativo (Linux, al menos) detecte el tamaño de la escritura y lo haga como se espera de todas maneras.

sección es de especial importancia para entender los resultados de este experimento *particular* con las implementaciones aquí descritas.

3.4. Árbol B (B-Tree)

Como ya se dijo en la introducción, un Árbol B almacena B elementos en cada nodo, y éste tiene $B + 1$ hijos. Cada hijo cumple con la misma propiedad, pero existen otros invariantes.

- Cada nodo, salvo la raíz, tendrá al menos $\lceil \frac{B}{2} \rceil$ elementos.
- Todas las hojas del árbol están a la misma profundidad.
- Todos los elementos de un nodo están ordenados.

Para mantener los invariantes es necesario explicar los algoritmos de inserción y búsqueda. En adelante K será el elemento buscado.

Búsqueda

Bastante simple, se compara K con todos los elementos de un nodo E_i hasta que $E_{>}K$ o bien $K = E_i$, si sucede esto último se encuentra el elemento y el algoritmo termina. Si no, se baja por el i-ésimo hijo del nodo. Nótese que es el i-ésimo, pues, en el momento que $E_i > K$, i corresponde al hijo que almacena los elementos más pequeños posibles que además son mayores que K.

Inserción

Se inserta un elemento en su posición correcta en la hoja que le corresponde y después empieza un proceso de *subir* por el árbol en que cada padre "arregla" a su hijo. Esto porque insertar un elemento en un nodo puede causarle *overflow*, de manera que si es que se admiten B elementos, no está permitido que un nodo tenga $B + 1$ elementos.

Así, un padre que detecta que un hijo suyo tiene overflow, toma el elemento ubicado en $\lceil \frac{B}{2} \rceil$, lo inserta en sí mismo, divide los elementos izquierdos y derechos en dos nuevos nodos, reagrupa sus los hijos de sus hijos, y espera a que su padre arregle su ahora potencial overflow.

Eliminación

Este es el proceso más complicado de todo, pues al eliminar se puede producir un *underflow*. Se divide el proceso de borrado en dos partes.

Bajada Si el elemento a borrar está en una hoja, se pasa directamente al proceso de subida. Si el elemento a borrar está en un nodo interno, se le pide el elemento de más a la izquierda al hijo derecho. Al llegar a una hoja, se le remueve dicho elemento y se inserta donde se borró el elemento en el nodo interno inicial. Después se pasa a la parte de Subida.

Subida Un nodo detecta que su hijo tiene underflow. Para ello encuentra un hermano (en la implementación se usa siempre el izquierdo, a menos que no se pueda) y dependiendo de cuántos elementos tenga es la acción que se debe tomar. Si el nodo hermano puede *prestarle* elementos al nodo con underflow, está todo bien y se ejecuta la acción que se denomina *shift*

Horario. Es decir, el hizo izquierdo pasa su elemento más a la derecha al padre y el padre pasa su elemento al hijo derecho (que tiene underflow), esto genera que el último hijo del hermano pasa a ser el primero del nodo con underflow. Si el hijo izquierdo tiene underflow se hace un proceso análogo que se llama *shift Antihorario*, en el que el hermano que pasa elementos es el derecho en vez del izquierdo.

Si no se puede hacer un shift, es porque ambos hermanos tienen exactamente $\lceil \frac{B}{2} \rceil$ elementos, y la operación que se debe hacer es un *merge*, que es exactamente inverso a la operación de dividir un nodo, esto es, el nuevo nodo tiene todos los elementos del nodo izquierdo, después el elemento del padre y por último todos los elementos del hijo derecho, con los hijos igualmente acomodados.

3.5. Hashing Extendible

El hashing extendible ocupa un poquito de memoria principal para su ejecución, a cambio la promesa es que garantiza $O(1)$ accesos a disco.

Se usa un *Trie* en memoria principal para subdividir los hashes de las strings según su k-ésimo bit de hash[9], los nodos internos parten el conjunto y las hojas apuntan a archivos en disco que contienen los elementos. Las búsquedas se hacen linealmente en las páginas. Esto garantiza exactamente 1 acceso a disco por búsqueda, y 3 por inserción.

Al insertarse un elemento se chequeda overflow en la página donde se inserta el elemento, de haberlo se cambia la hoja por un nodo interno y se rehashean todos los elementos de la página y se agrupan por su $(p+1)$-ésimo bit de su función de hash, en que p denota la profundidad de la hoja antigua. La eliminación es el proceso inverso, donde se vuelven a juntar dos hojas si es que ambas juntas almacenan B elementos.

3.6. Hashing Lineal

El hashing lineal requiere una función de hashing a la que se le pueda aplicar la operación `mod`. En cualquier estado del hash se almacena un entero S que cumple con la propiedad de que la estructura *quisiera* almacenar $2S$ páginas, y existe otro entero n (nótese la minúscula, para diferenciarlo de N, el input) que indica la cantidad de páginas reales que existen en la estructura, por ello antes se ha puesto en cursiva la palabra "quisiera".

Para buscar o insertar un elemento se sigue el siguiente algoritmo, sea b el bucket donde se debe buscar, y H el hash evaluado en el elemento a buscar:

1. $b = H \mod S$
2. si $bp < n \mod S$ entonces $b = H \mod 2S$

Al llenarse un bucket, se crea una lista de páginas correspondientes a un bucket donde están todos sus elementos. La política de expansión y contracción de la cantidad de páginas aceptadas es responsabilidad del usuario de la estructura. En este experimento sólo se usará

[9]En realidad hemos usado k bits de atrás hacia adelante, pero igualmente se puede hacer de adelante hacia atrás.

una política de expansión.

Se contraerá cuando la razón entre elementos totales y cantidad de buckets sea menor a 0.83, y se expanderá cuando la razón entre elementos totales y cantidad de buckets sea mayor a 1.5. Ambos factores están sobre la cantidad total de elementos por página.

3.7. Función de Hashing

Lo único que se ha hecho para la función de hashing es usar la representación binaria de las strings, es decir, a cada carácter se le asigna 2 bits que lo caracterizan, y con ello caen justamente 32 bits de hashing por cada cadena, esto cabe en un `unsigned int`. Note que todas las cadenas son de largo 15, o sea, 16 bytes en `C`, y por tanto 32 bits de hash, que equivalen a 4 bytes.

3.8. Páginas en memoria secundaria

Para almacenar los elementos correctamente en memoria secundaria lo que se ha hecho es *serializar* cada estructura de datos según el siguiente diseño:

Árbol B

- 4 bytes que corresponden a la cantidad de elementos de la página
- 4 bytes que corresponden a la cantidad de hijos de la página
- 4 bytes que corresponen al indice propio de esta página
- 3264 bytes que corresponen a 204 cadenas de 16 bytes.
- 820 bytes que corresponen a 205 índices posibles de cada hijo de la página

Correspondientes a 4096 bytes. Las páginas se almacenan en ese exacto órden. Por razones de tiempo y de complejidad, no se ha almacenado el par (K, V) asociado a un diccionario, sino que sólo K, esto no interfiere de ninguna manera en las pruebas ni accesos a disco, puesto que no se considera el procesamiento en memoria principal.

Hashing Extendible

Una página de hashing extendible tiene la siguiente estructura:

- 4 bytes que corresponden al número de elementos
- 4 bytes que corresponden a la profundidad de la página en el Trie.
- 4064 bytes que corresponden a 127 pares (K, V) de 32 bytes cada uno.

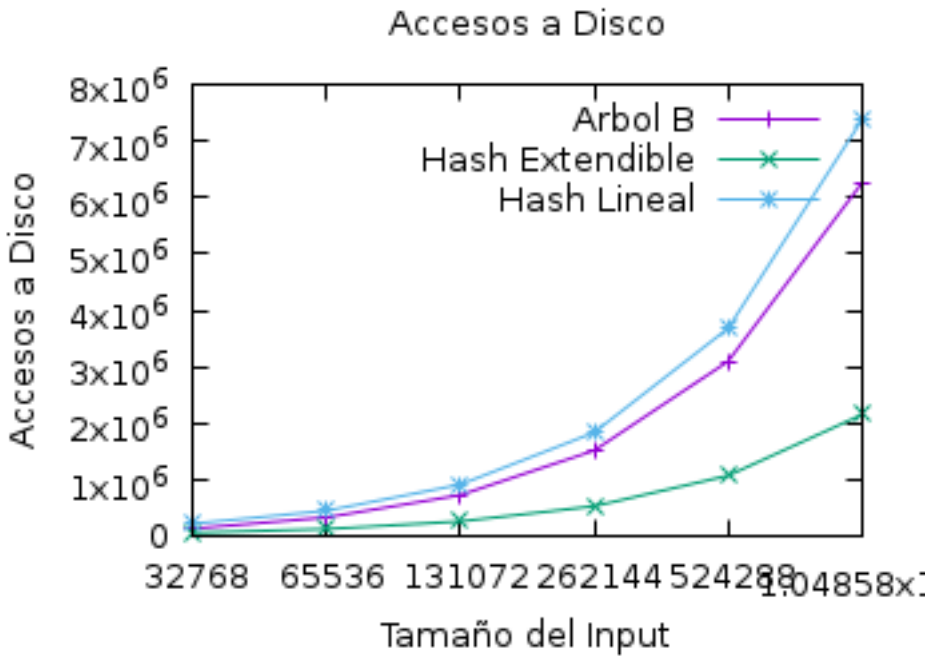
Esto da un total de 4072 bytes. Desafortunadamente no se puede almacenar más información de forma simple, así que se pierden 24 bytes por página.

Hashing Lineal

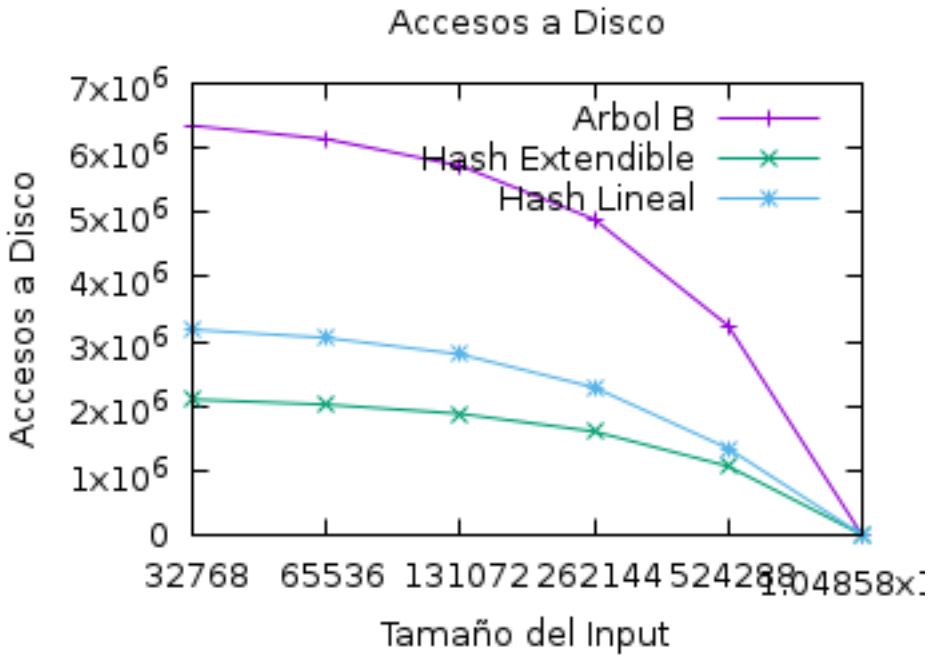
Una página de hashing lineal tiene casi la misma estructura que una página de hashing extendible, pero en vez de almacenar la profundidad de la página, se almacena el índice de la página en la lista de páginas de su propio bucket. Esto da también un total de 4072 bytes por página, 24 bytes perdidos y 127 pares de información relevante para el hash.

4. Resultados

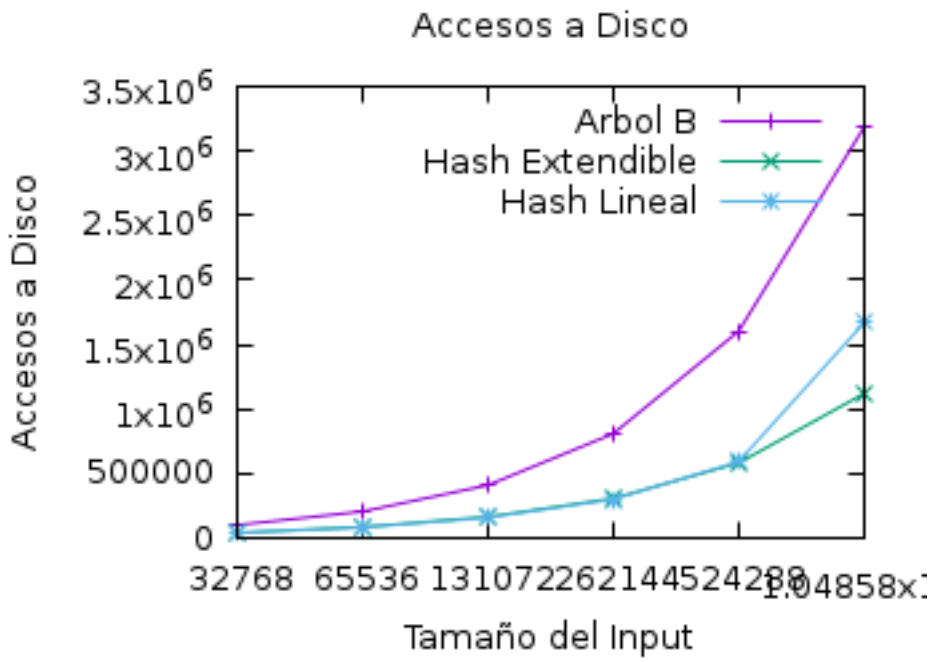
4.1. Inserción



4.2. Eliminación



4.3. Búsqueda


Accesos a Disco
Accesos a Disco
3.5x10^6
3x10^6
2.5x10^6
2x10^6
1.5x10^6
1x10^6
500000
0
Arbol B
Hash Extendible
Hash Lineal
32768
65536
Tamaño del Input

5. Análisis e Interpretación de los Datos

Se puede ver que en general, es mejor el Hashing extendible, pues realiza **siempre** menos accesos a disco que cualquier estructura. Queda pendiente el análisis de ocupación, que no se ha hecho por tiempo.

6. Anexos

6.1. Discusión del tiempo de ejecución y la precisión de las mediciones

Un análisis mas exhaustivo hubiese incluido una mayor cantidad de experimentos, de modo de disminuir el error asociado a las mediciones. En este experimento, por razones de tiempo, de sanidad mental, de calidad de vida y de estrés universitario no se ha hecho esta tan necesaria medición. Es más, se ha reducido el espacio de pruebas para cada estructura de datos desde 2^{25} a 2^{20}.

Siendo conscientes del hecho anterior, no es perdonable el hecho de no generar promedios, pero el tiempo de ejecución de un sólo experimento es demasiado grande como para esperar que se pueda entregar medidas aceptables. Lo que simplemente queda es esperar que la muestra generada al azar sea representativa de una condición probable de ejecución, que, dada por la naturaleza de repeticiones del experimento, estamos cerca del 63 %, pues