Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos

Práctica 2: Pruebas a Posteriori (Algoritmos de Búsqueda)

Equipo:
CompilandoConocimiento

Integrantes:
Morales López Laura Andrea
Ontiveros Salazar Alan Enrique
Rosas Hernández Óscar Andrés

Profesor:
Franco Martínez Edgardo Adrián

Índice

1. Introducción

Uno de los típicos problemas dentro de un curso de programación es la búsqueda. Estos algoritmos son la base de muchos otros, además de que tenemos con ellos unas ideas interesantes a usar en otro tipo de algoritmos, como búsqueda binarai y derivados, las estructuras de datos y los algoritmos concurrentes.

1.1. Métodos

1.2. Definición

1.3. Algoritmos de Búsqueda

2. Planteamiento del Problema

Con base en el ordenamiento obtenido a partir del archivo de entrada de la práctica 1 que tiene 10 millones de números diferentes; realizar la búsqueda de elementos bajo tres métodos de búsqueda, realizar el análisis teórico y experimental de las complejidades, así como encontrar las cotas de los algoritmos.

1. Búsqueda lineal o secuencial
2. Búsqueda binaria o dicotómica
3. Búsqueda en un árbol binario de búsqueda
4. Implementación de las tres búsquedas con Threads

3. Diseño de la Solución

4. Implementación de la Solución

4.1. Búsqueda Lineal

```
/*==============================================================================
====================           LINEAL SEARCH           ========================
==============================================================================*/
/**
 * Is just lineal search ...
 *
 * @param Data                 A pointer to the array of int to sort
 * @param DataSize             The size of the Data array
 * @param NumberToSearch       The number to search
 * @return                     Nothing...I'm modifying the raw data
 */
int LinealSearch(int Data[], int DataSize, int NumberToSearch) {     //=== LINEAL SEARCH =====

    for (int i = 0; i < DataSize; ++i) {                             //For each item in Data
        if (Data[i] == NumberToSearch)                               //If find it
            return i;                                                //Go
    }

    return -1;                                                       //Not found

}
```

4.2. Búsqueda Lineal Paralela

```c
/*=====================================================================
===========          PARALELL LINEAL SEARCH             ===============
=====================================================================*/

/**
 * Is just lineal search ... but I divide the search to N workers
 *
 * @param Data               A pointer to the array of int to sort
 * @param DataSize           The size of the Data array
 * @param NumberToSearch     The number to search
 * @return                   Nothing...I'm modifying the raw data
 */


/*=====================================================
=======      PARALELL AUXILIAR FUNCTIONS      =========
=====================================================*/

typedef struct LinealSearchDataStruct {                         //Parameters to the threads
    int Initial;                                                //Initial index to found
    int Final;                                                  //Final index to found
    int *Data;                                                  //Pointer to teh data
    int NumberToSearch;                                         //What I'm searching
    int *FoundIt;                                               //Flag
} LinealSearchData;

void* LinealSearchRange(void* Parameters) {                     //Thread Function
    LinealSearchData* Data = (LinealSearchData*) Parameters;    //Get the Parameters
    int Initial = Data->Initial;                                //Unwrap the data
    int Final = Data->Final;                                    //Unwrap the data

    for (int i = Initial; i<=Final && *Data->FoundIt==-1; ++i){ //For each item in Data
        if (Data->Data[i] == Data->NumberToSearch){             //If find it
            *Data->FoundIt = i;                                 //Now we have an index :)
            break;
        }
    }

    return NULL;                                                //I found nothing :(
}


/*=====================================================
=======        PARALELL MAIN FUNCTIONS        =========
=====================================================*/
int ParalellLinealSearch
    (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'LINEAL' SEARCH ======

    pthread_t* Workers =
        (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));      //Now get the array worker

    LinealSearchData* Parameters = (LinealSearchData*)
                malloc(NumOfWorkers*sizeof(LinealSearchData));      //Now create the parameters data

    int SizeOfSearch = DataSize / NumOfWorkers;                     //Get the size of the search

    int FoundIt = -1;                                               //Found it flags
    for (int i = 0; i < NumOfWorkers; ++i) {                        //For each worker:

        Parameters[i].Initial = i * SizeOfSearch;                   //Get the initial index to search
        Parameters[i].Final = (i == NumOfWorkers - 1)?
            (i + 1) * SizeOfSearch - 1: DataSize - 1;               //Get the final index to search

        Parameters[i].Data = Data;                                  //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        Parameters[i].FoundIt = &FoundIt;                           //Put the data

        pthread_create
            (&Workers[i], NULL, LinealSearchRange, &Parameters[i]); //Get the thread working!
    }

    for (int i = 0; i < NumOfWorkers; ++i)                          //For each worker
        pthread_join(Workers[i], NULL);                             //Now wait to the worker

    return FoundIt;                                                 //Return the result

}
```

4.3. Búsqueda Binaria

```
/*=============================================================================
=====================            BINARY SEARCH            =====================
=============================================================================*/
/**
 * Is just binary search ...
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */

int BinarySearch(int Data[], int DataSize, int NumberToSearch) {     //=== BINARY SEARCH =====
    int Initial = 0,  Final = DataSize;                              //Variables that we need

    while (Initial <= Final) {                                       //While find make sense

        int Middle = Initial + ((Final - Initial) / 2);              //Find a new SearchPosition

        if (Data[Middle] == NumberToSearch)                          //If all ok!
            return Middle;                                           //If we find it!
        else if (Data[Middle] > NumberToSearch)                      //If we need to go to side
            Final = Middle - 1;                                      //Find the new final position
        else                                                         //If we need to go to side
            Initial = Middle + 1;                                    //Find the new initial position
    }

    return -1;
}
```

4.4. Búsqueda Binaria Paralela

```
/*=====================================================================
=========        PARALELL AUXILIAR FUNCTIONS        ===========
=====================================================================*/
typedef struct BinarySearchDataStruct {                              //Parameters to the threads
    int Initial;                                                     //Initial index to found
    int Final;                                                       //Final index to found
    int *Data;                                                       //Pointer to teh data
    int NumberToSearch;                                              //What I'm searching
    int *FoundIt;                                                    //Flag
} BinarySearchData;

void* BinarySearchRange(void* Parameters) {                          //Thread Function
    BinarySearchData* Data = (BinarySearchData*) Parameters;         //Get the Parameters
    int Initial = Data->Initial;                                     //Unwrap the data
    int Final = Data->Final;                                         //Unwrap the data
    int Middle;

    while(Initial <= Final && *Data->FoundIt==-1){

        Middle = Initial + ((Final - Initial) / 2);                  //Find a new SearchPosition

        if (Data->Data[Middle] == Data->NumberToSearch){             //If we find it!
            *Data->FoundIt = Middle;
            break;
        }

        if (Data->Data[Middle] > Data->NumberToSearch)               //If we need to go to side
            Final = Middle - 1;                                      //Find the new final position
        else                                                         //If we need to go to side
            Initial = Middle + 1;                                    //Find the new initial position
    }

    return NULL;                                                     //I found nothing :(
}


/*=====================================================================
=========        PARALELL MAIN FUNCTIONS        ===========
=====================================================================*/
int ParalellBinarySearch
    (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'BINARY' SEARCH ======

    pthread_t* Workers =
        (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));       //Now get the array worker

    BinarySearchData* Parameters = (BinarySearchData*)
        malloc(NumOfWorkers*sizeof(BinarySearchData));               //Now create the parameters data

    int SizeOfSearch = DataSize / NumOfWorkers;                      //Get the size of the search

    int FoundIt = -1;                                                //Found it flags
    for (int i = 0; i < NumOfWorkers; ++i) {                         //For each worker:

        Parameters[i].Initial = i * SizeOfSearch;                    //Get the initial index to search
        Parameters[i].Final = (i == NumOfWorkers - 1)?
            (i + 1) * SizeOfSearch - 1: DataSize - 1;                //Get the final index to search

        Parameters[i].Data = Data;                                   //Put the data
        Parameters[i].NumberToSearch = ToSearch;                     //Put the data
        Parameters[i].NumberToSearch = ToSearch;                     //Put the data
        Parameters[i].FoundIt = &FoundIt;                            //Put the data

        pthread_create
            (&Workers[i], NULL, BinarySearchRange, &Parameters[i]); //Get the thread working!
    }

    for (int i = 0; i < NumOfWorkers; ++i)                           //For each worker
        pthread_join(Workers[i], NULL);                              //Now wait to the worker

    return FoundIt;                                                  //Return the result

}
```

4.5. Búsqueda con BTS

```
/*=====================================================================
=====================              BST SEARCH          ================
=====================================================================*/
/**
 * Just search like in a BTS, note that a node save the number and
 * the index of the number in the original array
 *
 * @param Tree       A pointer to a node; the root of a tree
 * @param ToSearch   The number to search
 * @return           The index of the number in the Original Array
 */
int SearchWithBST(Node* Tree, int ToSearch) {                        //=== BST SEACH =====
    Node **NewNode = &Tree;                                          //Let start at root

    while (*NewNode != NULL) {                                       //While are not at a leaf

        if ((*NewNode)->NodeItem == ToSearch)                        //If we found it!
            return (*NewNode)->Index;                                //return the index

        NewNode = (ToSearch < (*NewNode)->NodeItem)?                 //We have to move right
            &((*NewNode)->Left): &((*NewNode)->Right);               //Move left or right
    }

    return -1;                                                       //Not found!
}
```

5. Actividades y Pruebas

5.1. Comparativas Individuales

5.2. Comparativas Globales

5.3. Preguntas

- **¿Cuál de los 3 algoritmos es más fácil de implementar?**

  El más sencillo (o en algoritmia conocido coloquialmente como la bruta) es la búsqueda lineal. Es la solución más sencilla con apenas 3 o 4 líneas de código.

  Además de ser sencilla de implementar es en la que puedes cometer un error con menor probabilidad a la hora de definir los índices o variables de apoyo.

- **¿Cuál de los 3 algoritmos es el más difícil de implementar?**

  El más difícil de implementar en la búsqueda usando un Binary Search Tree, porque nos obliga a diseñar algoritmos que son comúnmente recursivos en algo iterativo.

- **¿Cuál de los 3 algoritmos es el más difícil de implementar en su variante con hilos?**

  Curiosamente la respuesta creo que tiene que ir para la versión con hilos de la búsqueda lineal.

  Por un lado el trabajar con hilos siempre complica las cosas, sobretodo porque involucra problemas de sincronicidad y el manejo de variables que pueden ser posiblemente accedidas por varios hilos. Aquí hay una consideración importante que hay que hacer: Solo estamos buscando una variable, por lo que no me tengo que preocupar con la memoria: el primer hilo que me mande una respuesta, esa la tomaré y todos los demás hilos al notar el cambio en la variable se matan.

- **¿Cuál de los 3 algoritmos en su variante con hilos resulto ser mas rápido?**

  La búsqueda binaria, es cierto, que fue más rápida la versión lineal, pero aún así la versión con hilos es endemoniadamente rápida.

- **¿Cuál algoritmo tiene menor complejidad temporal?**

  La búsqueda binaria. Es uno de los algoritmos más famosos por tener un $O(\log_2(n))$ aunque en ciertos casos, CON UN ÁRBOL BALANCEADO la búsqueda con árbol puede tener esa complejidad.

- **¿Cuál algoritmo tiene mayor complejidad temporal?**

  Sin duda alguna ese premio tiene que ir para la búsqueda lineal, en el peor de los casos tenemos un búsqueda de $O(n)$.

- **¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?**

  Sí, al menos en rangos generales de los algoritmos se esperaba y se ve la gran diferencia entre los algoritmos de búsqueda rápidos y la búsqueda lineal.

  Por otro lado a simple vista uno puede pensar que una variante con hilos siempre será más rápida que una versión que no es concurrente, pero esto no tiene porque ser así, por ejemplo en la búsqueda binaria es más que obvio que como solo estamos haciendo serie de comparaciones entonces el añadir hilos no mejora nada y solo sobrecarga al sistema operativo con su creación.

- **¿Sus resultados experimentales difieren mucho de los análisis teóricos que realizó? ¿A que se debe?**

  No, como continuación de la respuesta anterior, si se analizaban con algo de tranquilidad los algoritmos se podía llegar a unas muy buenas estimaciones.

- **¿Los resultados experimentales de las implementación con hilos de los algoritmos realmente tardaron $\frac{F(t)}{\#hilos}$ de su implementación sin hilos?**

  Jajaja, si hablamos de la búsqueda lineal, claro, es decir, no fue exactamente esa cantidad sobretodo por los grandes márgenes de error que tenemos a la hora de medir tiempos tan pequeños pero el punto es que sí, casi casi tenemos una mejora linealmente correspondiente con la cantidad de hilos, después de todo dividimos el trabajo entre la cantidad de hilos.

  Pero en los otros dos al ser escencialemente una gran cantidad de comparaciones en lineal entonces no podemos llegar a prácticamente ninguna mejora, incluso un desempeño un poco peor.

  En general, para n hilos tenemos una nueva función complejidad:

  - Búsqueda Lineal: $f_h(t) = \frac{f(t)}{\text{Número de hilos}}$
  - Búsqueda Binaria: $f_h(t) = 1f(t)$
  - Búsqueda BST: $f_h(t) = 1f(t)$

- **¿Cuál es el % de mejora que tiene cada uno de los algoritmos en su variante con hilos? ¿Es lo que esperabas? ¿Por qué?**

- **¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?**

  Sí, usamos una PC con las siguientes características:

  - HP EliteDesk 700 G1 SFF
  - 8GB de memoria RAM, DDR3 SDRAM non-ECC, 1600MHz
  - Procesador Intel(R) Core(TM) i5-4590 (4° generación), CPU @ 3.30GHz (4 CPUs), ∼3.3GHz. Intel vPro Technology
  - 1TB de disco duro HDD, Serial ATA-600 6Gb/s, 7200 rpm

  Y el software fue el siguiente:

  - Sistema operativo Linux
  - Distribución Elementary OS 0.4
  - Compliador GCC con soporte para C11
  - Python 3.6 con las bibliotecas `matplotlib` y `numpy`
  - No había ninguna aplicación abierta al momento de ejecutar los algoritmos

- **Si solo se realizara el análisis teórico de un algoritmo antes de implementarlo, ¿podrías asegurar cual es el mejor?**

  Con un análisis básico, es decir, con el uso de cotas, rápidamente podríamos descartar a la búsqueda lineal SI ES QUE NUESTRO ARREGLO ESTUVIERA ORDENADO; y ya que el BST requiere otro tiempo inicial para la creación del árbol, sería fácil decantarse por la sencilla búsqueda binaria.

  Pero si tenemos un arreglo que no está ordenado las cosas cambian, ahí depende mucho del tamaño del problema y de la cantidad de hilos que podemos correr eficientemente lo que decanta la balanza por BST o un búsqueda lineal.

- **¿Qué tan difícil fue realizar el análisis teórico de cada algoritmo?**

  - Búsqueda Lineal: Sencillo, de esos que son ejemplos básicos en clase.
  - Búsqueda Binaria: Algo más compleja, sobretodo por el cálculo de los valores límites, pero la verdad es que no hay mucho más allá.
  - Búsqueda BST: Igualmente, el peor caso en un árbol BALANCEADO es bastante sencillo de obtener.

- **¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?**

  - Hagan sus scripts jóvenes, pues aunque parezca más tedioso y crean que es más rápido anotar toda la información solicitada (que es mucha) a mano, es más eficiente por si se requiere cambiar algún algoritmo o los valores de entrada. Al menos que el script guarde los tiempos en algún archivo de texto con un formato que quieran.
  - Prueben sus algoritmos con arreglos pequeños antes de buscar en los 10 millones para ver si realmente los programaron bien.
  - Hagan sus programas generales, es decir, que reciban cualquier tamaño de subarreglo y el algoritmo a usar.
  - Usen Linux, pues en Windows no están las bibliotecas para medir los tres tiempos.
  - Cuando corran sus algoritmos, procuren cerrar todas las tareas en segundo plano para que los tiempos obtenidos sean lo más apegados a la realidad posible.

6. Errores Detectados

Por un lado, no hemos detectado errores al momento de correr los algoritmos que no hallamos sabido como resolver, bueno, si, solo uno.

El Tiempo.

A la hora de medir el tiempo de los algoritmos tenemos un grave problema, y es que a la hora de usar las funciones estandar proporcionadas por el profesor para medir el tiempo tenemos que sus mediciciones son incorrectas en el sentido de que sabemos que la ecuación para calcular el porcentaje de uso de la CPU esta dada por:

$$CPUWall = (UserTime + SysTime)/RealTime$$

Pero al momento de intentar medir este porcentaje con búsquedas ridiculamente rápida tenemos que nos dan valores sobre el 100 % lo que indica que las mediciones de tiempo son impresisas si trabajamos con intervalos pequeños de tiempo

Por otro lado no "errores" en si, pero cosas que podrían ocasionar un error en los programas son:

- Que no se sigan las indicaciones de los parametros de consola, es decir:
  - Darme mas casos de los que tengo en el archivo
  - Darme mas tamaño para el arreglo que números
  - Darme rutas incorrectas a la hora de abrir los ficheros
- Que no se ingresen los parametros de consola
- Que no se tenga configurada la versión de python3.6 sino la 3.5 que es la estandar en este momento
- Que no existan los directorios donde estarán las salidas y las gráficas

7. Posibles Mejoras

8. Conclusiones

- **Alan:**
- **Laura:**
- **Óscar:**