ESTRUCTURA DE DATOS Y DE ALGORITMOS

Tarea 1: Análisis de algoritmos y tiempo de procesamiento

Carrera:

Ingeniería civil en computación

Profesor:

José M. Saavedra Rondo

Integrantes:

Renato Aguirre

Diego Llull

Agosto 25, 2024

Resumen

En este informe se enfoca en estudiar, comprender dos algoritmos de búsqueda para determinar la presencia de palabras de un conjunto de consultas en un diccionario. Se utilizan 2 algoritmos clave para esto el Algoritmo Secuencial (AS) y el Algoritmo de Búsqueda Binaria (ABB), el que debe estar ordenado mediante un Sorteo para ser ejecutado, los dos son evaluados sus desempeños en diferentes tamaños de diccionarios.

Se describe en detalle el diseño de cada algoritmo, en donde se destacan las distintas estrategias que emplean para abordar el problema y resolverlo.

Los resultados en base a tablas de tiempo y a gráficos revelan como cada algoritmo emplea una diferente estrategia, y como el uso de distintas estructuras de datos nos permiten lograr tiempos distintos

Introducción

El diseño eficiente de algoritmos es fundamental en el campo de la informática, ya que la elección de un algoritmo adecuado puede influir significativamente en el rendimiento y la escalabilidad de una aplicación. Esta tarea se enfoca en analizar el tiempo de ejecución de dos algoritmos de búsqueda diferentes: el Algoritmo Secuencial (AS) y el Algoritmo de Búsqueda Binaria (ABB). El AS realiza una búsqueda lineal, revisando cada término del diccionario uno por uno, mientras que el ABB, más eficiente en su complejidad temporal O(logN), requiere que el diccionario esté previamente ordenado. Para poder ordenarlo, se necesita un algoritmo de ordenación, en esta ocasión se usará, MergeSort.

Para esta tarea, se establecieron las siguientes condiciones experimentales: el número de consultas Nq es fijo e igual a 1 millón, y el tamaño del diccionario Nd varía entre 10K, 50K, 100K, 200K y 400K palabras. Para cada tamaño de diccionario, se estimará el tiempo promedio de búsqueda de los términos utilizando ambos algoritmos. Además, se evaluará el ABB considerando tanto el tiempo de búsqueda como el tiempo necesario para ordenar el diccionario previamente. Los resultados se presentan mediante tablas y gráficos, que ilustran cómo el rendimiento de cada algoritmo varía con el tamaño del diccionario y cómo se comparan entre sí en términos de eficiencia.

El objetivo es determinar las palabras en común entre las consultas y el diccionario, incrementando progresivamente la cantidad de palabras utilizadas y observando cómo esto afecta al tiempo de ejecución de cada algoritmo, lo que se refleja en las tablas y gráficos incluidos en el análisis.

Desarrollo

Para llevar a cabo los experimentos, se desarrolló un programa en C ++ que implementa los algoritmos de búsqueda.

Este programa vive en un [repositorio de github]donde está todo el código detallado.

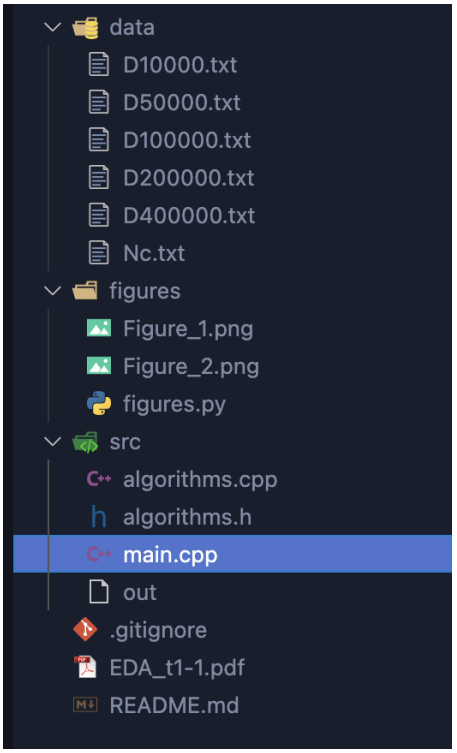


fig. 1 Directorio base del código utilizado}

El directorio base contiene 3 carpetas, data, figures y src. En la primera, están todos los diccionarios y el archivo de consultas (descargados desde el enlace del enunciado), en la segunda carpeta (figures) está el código de python utilizado para los gráficos. En la carpeta src se encuentra el código en C ++ con los algoritmos y el archivo main del programa.

Algoritmo Secuencial (AS) y el Algoritmo de Búsqueda Binaria (ABB).

```
int string_in_array_secuential_search(std::string array[], int size, std::string key)
{
  for (int i = 0; i < size; i++)
  {
    if (array[i] == key)
    {
      return i;
    }
  }
  return -1;
}
```

fig. 2 algoritmo de búsqueda secuencial implementado. Elaboración propia

El algoritmo de búsqueda secuencial funciona haciendo un loop sobre los elementos del array diccionario, y comparando cada elemento con la palabra buscada, en caso de encontrarla, retorna el índice de esta palabra, en caso contrario, retorna 1.

Este algoritmo tiene una complejidad teórica de O(n)

```
int string_in_array_binary_search_recursive(std::string sorted_array[], int low, int size, std::string key)
{
  if (size < low) // string not found
  {
    return -1;
  }
  int half = low + ((size - low) / 2);
  //std::cout << "half: " << half << std::endl;
  if (key.compare(sorted_array[half]) == 0) // base case
  {
    return half;
  }
  else if (key.compare(sorted_array[half]) < 0) // string is less than the middle element, ignore second half
  {
    return string_in_array_binary_search_recursive(sorted_array, low, half - 1, key);
  }
  else  // string is bigger, ignore the first half
  {
    return string_in_array_binary_search_recursive(sorted_array, half + 1 , size, key);
  };
  return -1;
}
```

fig. 3 algoritmo de búsqueda binario implementado. Elaboración propia

El algoritmo de búsqueda binario (ABB) funciona eligiendo la palabra en el medio del array, si esta es la palabra que se busca, se retorna el índice, si no, se compara.

Si la palabra que buscas va antes en el abecedario que la palabra en el medio, descartas la mitad derecha de la lista (todas las palabras que van después que esta).

Si la palabra que buscas va después que la palabra en el medio, descartas la mitad izquierda de la lista (todas las palabras que van antes).

Para este programa se implementó una versión recursiva de este algoritmo.

Este algoritmo tiene una complejidad teórica de O(log(n)).

Cabe mencionar que para poder utilizar el ABB se requiere tener un diccionario ordenado. Para poder ordenar el diccionario de manera correcta y eficiente, se eligió el algoritmo de ordenación "*Merge Sort*", tiene una complejidad de O(N log(N)). Se eligió este Algoritmo de ordenación debido a que su tiempo de ejecución es eficiente, es estable y funciona muy bien en arrays grandes, además tiene un tiempo de ejecución más consistente en comparación al QuickSort por ejemplo.

Luego, en el programa main, simplemente se busca cada palabra en la lista de consultas (para cada combinación de diccionario y algoritmo) y se mide el tiempo en el cual se demoró en encontrar todas las consultas para cada algoritmo.

En el caso de ABB también se mide el tiempo en el cual se demora en ordenar el diccionario por separado

```
void run_and_time_binary_search(const int DICTIONARY_SIZE, std::string query_array[])
{
  /* get dictionary */

  std::string* dictionary_array = new std::string[DICTIONARY_SIZE];
  const std::string filename = "../data/D" + std::to_string(DICTIONARY_SIZE) + ".txt";
  //std::cout << "reading from file: " << filename << std::endl;
  read_file_and_insert_into_array(filename, dictionary_array, DICTIONARY_SIZE);
  //std::cout << "Dictionary read" << std::endl;

  /* sort and time dictionary array*/
  std::cout << "Sorting dictionary" << std::to_string(DICTIONARY_SIZE) << std::endl;
  auto sort_start = std::chrono::high_resolution_clock::now();

  std::sort(dictionary_array, dictionary_array + DICTIONARY_SIZE);

  auto sort_end = std::chrono::high_resolution_clock::now();
  auto sort_duration = std::chrono::duration_cast<std::chrono::milliseconds>(sort_end - sort_start);
  std::cout << "Dictionary sorted" << std::endl;
  std::cout << "Sort time: " << sort_duration.count() << " milliseconds" << std::endl;

  /* search queries and time it */
  std::cout << "Searching dictionary" << std::to_string(DICTIONARY_SIZE) << std::endl;
  auto start = std::chrono::high_resolution_clock::now();
  search_in_dictionary_binary(dictionary_array, query_array, DICTIONARY_SIZE);

  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
  std::cout << "search done in " << duration.count() << " milliseconds" << std::endl;

  delete[] dictionary_array;
}
```

Fig 4. función para buscar todas las consultas en un determinado diccionario.

Elaboración propia.

Finalmente el programa muestra en consola en cual paso de la ejecución va y el tiempo demorado por diccionario y por algoritmo. Además muestra la cantidad de palabras encontradas y no encontradas por cada algoritmo.

Para más detalle ver el código fuente del programa.

Ejemplo de ejecución

Para compilar el programa: (macOS y linux) es necesario estar en el directorio src.

$ cd src

$ g++ -o out main.cpp algorithms.cpp -std=c++11

Para correr el programa:

$ ./out

```
(base) ➜  src git:(main) ✗ ./out
Reading queries
Queries read
Setup done.

Running binary search with 10.000 words
Sorting dictionary10000
Dictionary sorted
Sort time: 8 milliseconds
Searching dictionary10000
Binary search: 140 found, 9860 not found
search done in 16 milliseconds


Running sequential search with 10.000 words
reading from file: ../data/D10000.txt
Dictionary read
Sequential search: 140 found, 9860 not found
Sequential search done in: 1837 milliseconds
```

fig. 5 Extracto del output del programa en ejecución. Elaboración propia.

En la sección de figuras, se encuentra el script en python para obtener los gráficos. Este programa recibe los datos previamente extraídos del output del programa main y muestra los gráficos a través de la librería matplotlib.

```python
data = {
    "binary_search": [16, 87, 186, 404, 865],
    "sort": [
        8,
        47,
        102,
        246117,
    ],
    "binnary_search_plus_sort": [24, 134, 288, 621, 1326],
    "sequential_search": [1837, 43587, 167108, 620368, 2062273],
    "n": [10000, 50000, 100000, 200000, 400000],
}
data_without_sequential_search = {
    "binary_search": [16, 87, 186, 404, 865],
    "sort": [8, 47, 102, 217, 461],
    "binnary_search_plus_sort": [24, 134, 288, 621, 1326],
    "n": [10000, 50000, 100000, 200000, 400000],
}

# figure 1
plt.plot(data["n"], data["binnary_search_plus_sort"], label="binnary_search_plus_sort")
plt.plot(data["n"], data["sequential_search"], label="sequential_search")
plt.legend()
plt.xlabel("n")
plt.ylabel("time (ms)")
plt.title("Binary search vs Sequential search")
plt.show()
```

Fig.6 script en python para graficar. elaboración propia

Estos gráficos serán vistos en la parte de resultados.

Para correr este script asegúrese de tener instalado python y matplotlib.

$ python figures.py

Resultados experimentales y discusión

La máquina que utilizamos para el experimento es una Macbook Air que contiene un procesador M1 de 8 núcleos 4 de rendimiento y 4 de eficiencia, también cuenta con GPU M1 con 8 núcleos y también 8gb de memoria Ram.

A continuación, se presentarán los resultados para cada método de solución.

	Tiempo de ejecución. (ms)		
Tamaño (ND)	**AS**	**ABB**	**ABB + Sort**
10K	1837	16	24
50k	43587	87	134
100k	167108	186	288
200k	620368	404	621
400k	2062273	865	1326

FIg.7 Tabla de resultados. Elaboración propia

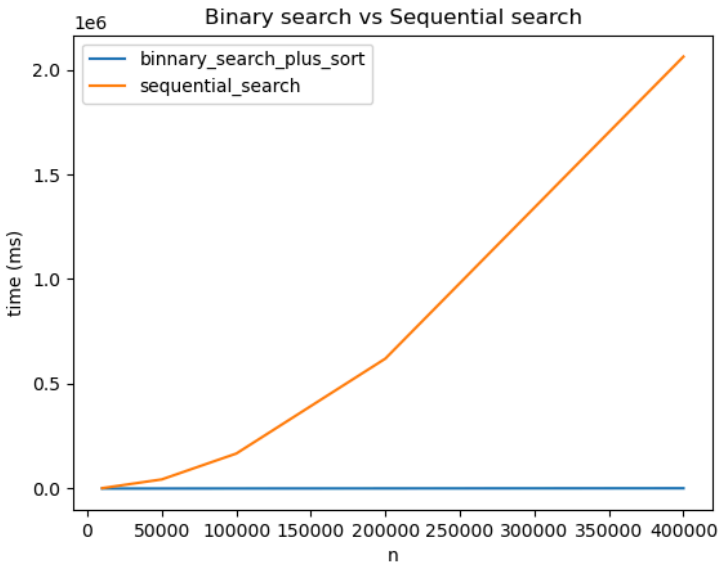


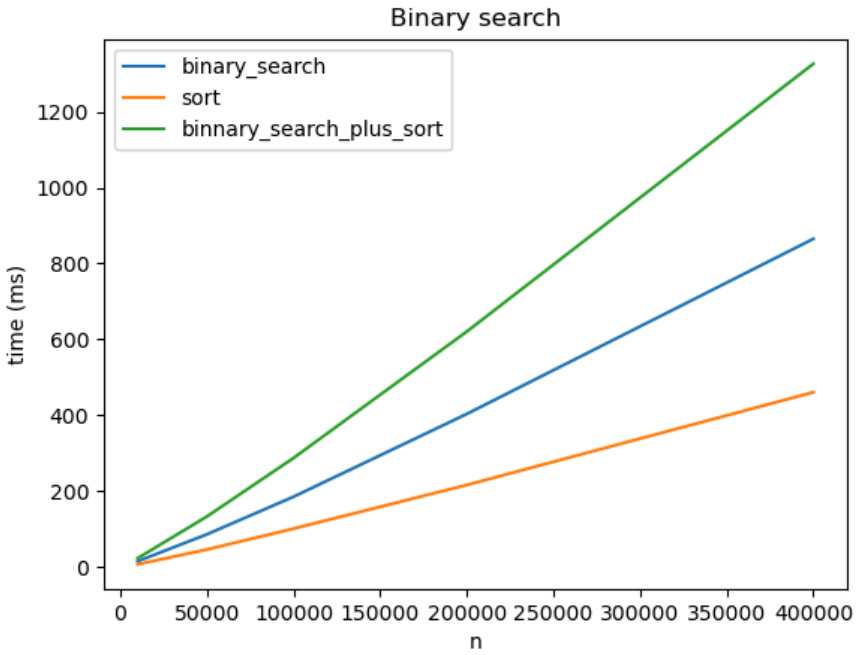
Fig. 8 ABB + sort vs AS. Elaboración propia.

Como se puede observar AS demoró órdenes de magnitud más que ABB.

con su valores variando desde 1,8 s hasta 2.062.273 s,

Al comparar el tiempo de AS teórico con el obtenido podemos observar que es aproximadamente igual,

tenemos que los 10k datos se demoraron 1837 ms en completar, lo que nos da aproximadamente 1 instrucción cada 5 ms ó 5 ms/instrucción. Al multiplicar estos 5 ms por los 400 mil datos nos da aproximadamente 2 millones de ms, que en efecto fue *aproximadamente* lo que demoró nuestro algoritmo.



Conclusión

En este informe se centró en analizar los tiempos de rendimiento del Algoritmo secuencia(AS) y el Algoritmo de Búsqueda Binaria(ABB), dos algoritmos de búsqueda. El objetivo es determinar las palabras en común entre las consultas y el diccionario, aumentando el número de palabras en el diccionario.

Al terminar este experimento se demostró que el Algoritmo de Búsqueda Binaria superó enormemente al Algoritmo Secuencial en cuanto a tiempos y esta diferencia se iba acrecentando mientras más grande era el diccionario. Y esto se debe principalmente a la complejidad de los algoritmos debido a que AS es O(N), lo que significa qué busca de uno en uno, es decir, el tiempo aumentará según el tamaño de los datos, mientras que el ABB tiene una complejidad de O(LOG N), lo que significa que aumentará lentamente, lo que lo hace más eficiente cuando hay mayor cantidad de datos. Sin embargo, el ABB requiere que el diccionario esté ordenado, para este experimentó se utilizó específicamente el MergeSORT, se eligió este algoritmo debido a su eficiencia, estabilidad y consistencia, lo porsupuesto que le añade una complejidad quedando en O(N LOG N).

En síntesis, el ABB + SORT sigue siendo por lejos más eficiente que el Algoritmo Secuencial, debido a que como se puede ver que con 10k de palabras el AS demoró 1837(ms) y el ABB +SORT demoró 24(ms), y la diferencia se hacia mas grande mientras más palabras.