Informe No. 1 Algoritmos de ordenamiento y multiplicación de matrices

Doctorado en ciencias de la computación

Facultad de Ingeniería, Universidad de Concepción

Estudiante: Carlos Alejandro Peña Varas

Introducción

En el presente informe se reporta cómo se realizó una evaluación experimental de diferentes algoritmos de ordenamiento y multiplicación de matrices. Para cada caso, se programó o se buscó algoritmos disponibles en internet, se generaron datasets de diferentes tamaños n, se ejecutaron los algoritmos con estos diferentes datasets calculando los tiempos de ejecución, se graficaron los tiempos para cada algoritmo y se compararon los resultados experimentales con los teóricos que se esperaban de acuerdo con la complejidad de cada algoritmo.

Algoritmos

Algoritmos de ordenamiento

Todos los algoritmos de ordenamiento se programaron en el mismo programa de C ++ llamado sorting.cpp, en la carpeta I1_Ordenamiento, al que se le deben dar un archivo que contengan primero un número del 1 al 4 que representa la opción de ordenamiento a elegir: 1 mergesort, 2 quicksort, 3 bubblesort y 4 algoritmo estándar, seguido de la lista de elementos a ordenar, para ver ejemplos revisar la carpeta datasets. Adicionalmente a cada algoritmo se lo llama como archivos de cabecera separados, excepto por el algoritmo estándar de C++ que se lo llama usando la libreria “algorithm”.

Librerias usadas:

- iostream: Proporciona herramientas para leer y escribir datos en la entrada y salida desde el teclado.
- algorithm: Contiene diversos algoritmos para realizar operaciones con datos, vectores y matrices, y diferentes operaciones como ordenar, buscar, eliminar elementos, etc.
- vector: Permite almacenar datos en una estructura de datos dinámica de manera secuencial en memoria.
- chrono: Proporciona herramientas para medir el tiempo de ejecución de un programa.

- fstream: Permite y escribir archivos.
- sstream: Permite trabajar con cadenas de caracteres y cambiar el tipo de dato.

Mergesort

Es un algoritmo de ordenamiento que usa el enfoque de “divide y conquista” en el que para ordenar una lista de elementos desordenados se divide la lista en la mitad y se ordena de forma recursiva cada sublista, finalmente se concatenan las listas resultantes en una nueva lista ordenada. Este algoritmo tiene una complejidad O(n log n).

Quicksort

Es un algoritmo de ordenamiento que también usa el enfoque de “divide y conquista” en el que para ordenar una lista de elementos desordenados se elige un elemento o pivote, en función del cual se reorganizan todos los demás elementos dejando a la izquierda los elementos que son más pequeños que el pivote y a la derecha los que son más grandes que él. Este proceso se ejecuta recursivamente en las siguientes sublistas (izquierda y derecha) hasta que el índice de comparación se cruza con el pivote. Este algoritmo tiene una complejidad de O (n log n) y O(n^2) en el peor de los casos, cuando la lista está semi-desordenada o si se elige un mal pivote, como el primer elemento de la lista.

Bubblesort

El algoritmo de ordenamiento Bubblesort se basa en comparar cada elemento con el siguiente y evaluar si el primero es mayor, si es mayor se intercambian los valores y se vuelve a comparar con el siguiente par de elementos hasta llegar al final de la lista. Su complejidad es O(n^2).

Algoritmo de ordenamiento estándar de C++

Este algoritmo está incluido en la biblioteca “algothim” de C++ y es una mezcla de Quicksort y Heapsort, en el que también se elige un pivote y se ordenan las sublistas dejando los valores menores al pivote a la izquierda y los mayores a la derecha, pero si detecta que la recursion es muy larga cambia a Heapsort, en el que toma el valor más grande y lo coloca al final de la lista Heap, luego repite el proceso con los elementos restantes hasta haber ordenado la lista. Tiene complejidad O(n log n) y O(n^2) en el peor de los casos al usar quicksort y elegir un mal pivote o tener una lista semi-desordenada.

Algoritmos de multiplicación de matrices

Los algoritmos de multiplicación de matrices iterativos cúbico tradicional y optimizado están en la carpeta I1_Multiplicación_de_matrices/Algoritmos_iterativos. para ejecutarlo se le debe dar un archivo de entrada en el que la primera línea tiene las dimensiones de las matrices seguido de las matrices, ver ejemplos en la carpeta 1_Multiplicación_de_matrices/Algoritmos_iterativos/datasets. Mientras que el algoritmo de strassen está en I1_Multiplicación_de_matrices/Strassen y para ejecutarlo hay que darle un archivo que tenga las matrices separadas una arriba de la otra, ver ejemplos en la carpeta I1_Multiplicación_de_matrices/Strassen/datasets

Algoritmo iterativo cúbico tradicional

Es el algoritmo tradicional de multiplicación de matrices, en el que cada iteración se realiza la multiplicación de una fila de la primera matriz con una columna de la segunda matriz, y se acumula el resultado en la posición correspondiente de la matriz resultante. Aunque es fácil de implementar, su complejidad temporal es O(n^3), lo que lo hace ineficiente para matrices grandes.

Algoritmo iterativo cúbico optimizado

El algoritmo iterativo cúbico optimizado para la multiplicación de matrices utiliza la optimización de bucle para mejorar el rendimiento del algoritmo iterativo cúbico tradicional. La técnica de optimización de bucle consiste en reorganizar el orden de los bucles anidados para minimizar los accesos a memoria y maximizar la reutilización de datos. Aunque la complejidad temporal sigue siendo O(n^3), la eficiencia del algoritmo es mayor que la del algoritmo iterativo cúbico tradicional para matrices grandes debido a la optimización de bucle.

Algoritmo de Strassen

El algoritmo de Strassen es un algoritmo de multiplicación de matrices que utiliza una técnica de "divide y conquista" para reducir la complejidad temporal de la multiplicación de matrices de O(n^3) a O(n^log2(7)), donde log2(7) es aproximadamente 2.81. La técnica de "divide y conquista" implica dividir las matrices de entrada en submatrices más pequeñas, resolver el problema de manera recursiva para las submatrices más pequeñas y combinar las soluciones para obtener la solución final. El algoritmo de Strassen utiliza esta técnica para reducir el número de multiplicaciones de matrices que se necesitan para calcular la multiplicación de matrices, de 8 en el algoritmo tradicional a 7 en el algoritmo de Strassen.

Datasets

Algoritmos de ordenamiento

Los datasets para los algoritmos de ordenamiento crearon usando un programa en c++ llamado datasets.cpp en las carpetas semiordenados y ordenados de I1 Ordenamiento/datasets, la diferencia es que el programa en semiordenados hace un ordenamiento estandar de la mitad de los datos. En cada carpeta se escriben sets de datos listas tamaño con n de 1 a 200.000 con incrementos de 20.000 con valores aleatorios.

Algoritmos de multiplicación de matrices

En el caso de los algoritmos iterativos encontrado en se usaron sets de datos generados arbitrariamente con matrices de diferentes tamaños 1_Multiplicación_de_matrices/Algoritmos_iterativos/datasets.

Mientras que en el caso de Strassen se generaron sets de datos usando un script en python que genera matrices de tamaño n cuadrado de 1 a 1000 con incrementos de 100. El script que genera las matrices se llama create_matrix.py ubicado en 1_Multiplicación_de_matrices/Strassen/datasets/cuadradas. También se generaron matrices no cuadradas usando las mismas matrices y reduciendo las columnas de la segunda matriz a 6 columnas de forma arbitraria por edición de archivos.

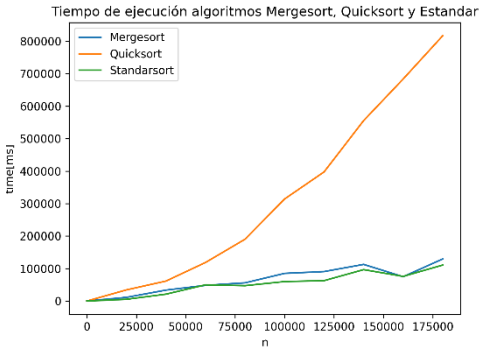
Resultados experimentales

Algoritmos de ordenamiento

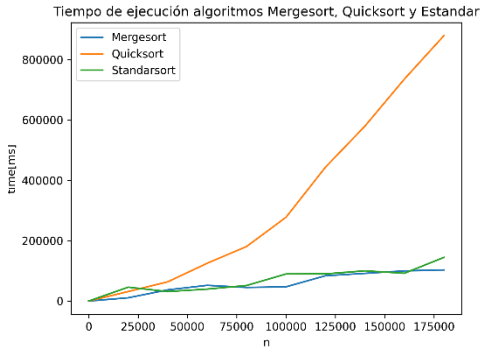
Para evaluar los algoritmos de ordenamiento se escribieron pequeños programas en bash (experimento.sh) en los que se les da al programa los archivos del dataset y el programa escribe un archivo CSV por cada algoritmo en el que se registraron el tamaño n de la lista a ordenar y el tiempo en microsegundos. Luego se graficaron los tiempos usando un script en python modificado del que nos entregó el ayudante Vicente en los laboratorios. Esto se hizo con los sets de datos desordenados y semi-desordenados

Como se puede ver en los siguientes gráficos el algoritmo con peor desempeño para n 200.000 fue el algoritmo Bubblesort, cosa que era de esperarse con complejidad cuadra. El siguiente peor es Quicksort que en el peor de los casos podría tener complejidad cuadrada, y esto se nota al ver que su rendimiento incluso peor con una lista desordenada.

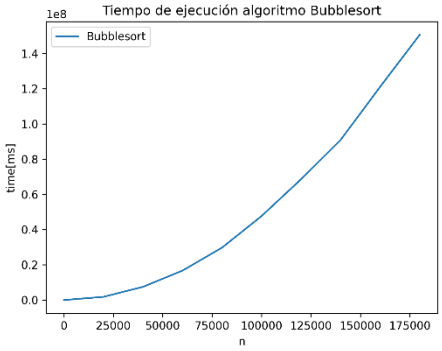
Elementos desordenados



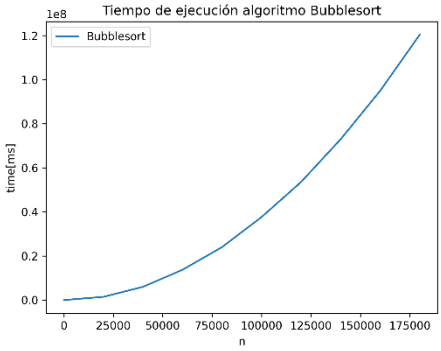
Elementos semidesordenados



Elementos desordenados



Elementos semidesordenados

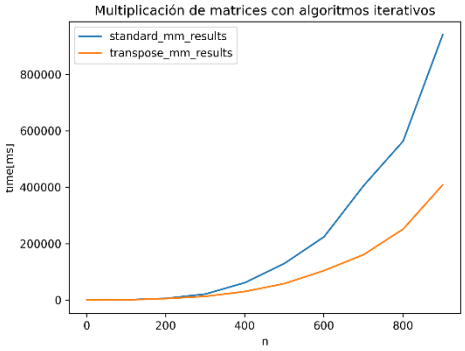


Algoritmos de multiplicación de matrices

Para los algoritmos de multiplicación de matrices se evaluaron de formas diferente, en el caso de los algoritmos iterativos se evaluó usando el archivo make proporcionado por el ayudante que corre el programa con los datasets usando el modo test del programa que varía n de 1 a 1000 con incrementos de 100. Para Strassen se usó un bash, como en el caso de algoritmos de ordenamiento, que evalúa el programa dando los datasets generados con matrices cuadradas y no cuadradas de tamaño n desde 1 a 1000 con incrementos de 100.

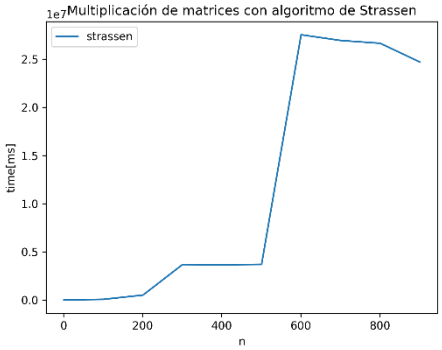
Los resultados muestran que el algoritmo optimizado tiene mejor rendimiento que el algoritmo tradicional de multiplicación de matrices. Esto era de esperar pues el algoritmo transforma las columnas de las segunda matriz a una fila, agilizando el acceso en memoria.

Por otro Lado el algoritmo de Strassen resultó ser el con el peor rendimiento con n 1000, esto podría explicarse por la cantidad de variables que se deben generar para ejecutar el algoritmo, pues estas deben escribir en memoria, siendo menos eficiente. La forma del gráfico sugiere que podría mejorar el rendimiento a mayor n, pero no pude comprobarlo.

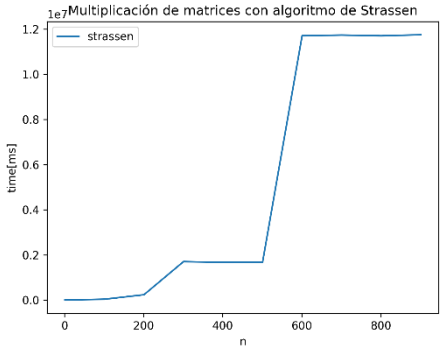


Algoritmo de Strassen:

Matrices cuadradas



Matrices no cuadradas



Conclusiones

Los algoritmos a pesar de tener complejidades que prometen mejorar sus rendimientos, pueden variar drásticamente al hacer experimentos reales, donde además de la naturaleza de los datos el hardware en que se ejecutan cumple un rol fundamental. A la hora de elegir un algoritmo se debería hacer un perfil de diferentes algoritmos que se acomoden tanto al tamaño de los datos a calcular como la arquitectura del hardware.