Trabajo Práctico 1

Recursividad

Fecha de entrega: 21 de noviembre, 2019

Materia: Algoritmos y Programación II (95.12)

Titular de cátedra: Ing. Patricia Calvo

Auxiliar: Ing. Leandro Santi

Auxiliar: Lic. Lucio Santi

Colaborador: Federico Geloso

Integrantes:

- Fernández Manoukian, Tomás (100773)
  tomymanoukian@gmail.com
- Mundani Vegega, Ezequiel (102312)
  ez.equidave@gmail.com
- Ramos, Federico (101640)
  fedearamos97@gmail.com

Diseño e implementación del programa

El programa diseñado consiste en la aplicación de la transformada de Fourier o su inversa utilizando distintos algoritmos a un conjunto de números. Se han programado los TDA vector y complejo, las funciones FFT, IFFT, DFT e IDFT y funciones que procesan argumentos por línea de comandos. Complementariamente se han hecho archivos de prueba con los que se pudo automatizar la comprobación del correcto funcionamiento del programa.

Dado el programa es una adaptación del utilizado para el "Trabajo Práctico 0: Programación en C++" que ya fue analizado y evaluado, se pondrá atención principalmente en analizar los algoritmos y funciones nuevas.

Se dividió el proceso del diseñado en tres partes. Una persona se encargó de la elaboración de las funciones FFT, IFFT y adaptar el procesamiento de argumentos por línea de comandos; otra de crear nuevos casos de prueba y automatizarlas; y otra de realizar los análisis temporal y espacial de los algoritmos. Se optó por utilizar GitHub para poder controlar grupalmente las versiones.

Al ejecutar el programa se pueden agregar los siguientes argumentos con sus versiones cortas: "--input", "--output", "--method" y "--help". En caso de no especificarse un archivo de entrada o salida se utilizan por defecto "cin" y "cout". En caso de no especificar el método se utiliza FFT por defecto.

Una vez que se procesaron los argumentos, se comienzan a leer todos los números complejos con el formato "(RE, IM)"o números reales sueltos de una línea y cada uno es cargado en un vector de complejos, ignorando espacios en blanco. El largo del vector es igual a la cantidad de caracteres de la línea dividido dos más uno, de esta manera no se tendrá que agrandar y el procesamiento será más rápido. Cuando se termina de leer la línea, se le aplica al conjunto de complejos la función que corresponda y se imprime por la salida elegida el vector transformado. Luego se libera el vector utilizado y se procesa la línea siguiente hasta terminar el archivo. En caso de haber un error, se ignora toda la línea, se imprime un error y se sigue leyendo el resto del archivo. Al finalizar e limpia el buffer y se cierran los archivos abiertos en caso de haberlos.

Complementariamente se han hecho programas que automatizan la comprobación del correcto funcionamiento del programa. Los valores de estas pruebas fueron calculados manualmente, y dado a que se trabaja con números irracionales, se estableció que si el resultado difería en menos de 10^{-6} era considerado correcto. Además se comprobó que no haya pérdidas de memoria con *valgrind*.

Algoritmos involucrados en la solución del trabajo

Se desarrollaron cuatro funciones principales: FFT, IFFT, DFT e IDFT. Todas fueron desarrolladas en base a sus algoritmos correspondientes.

DFT: Discrete Fourier Transform

Este algoritmo permite calcular la transformación de una secuencia de *N* puntos del tipo *x(0), ... , x(N − 1)* dada por la fórmula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}; k = 0, \ldots, N-1 \quad (1)$$

$$; W_N = e^{-j\frac{2\pi}{N}}$$

Su complejidad temporal pertenece a $O(n^2)$ y la espacial a O(n).

IDFT: Inverse Discrete Fourier Transform

Este algoritmo realiza como su nombre lo indica, la operación inversa: permite obtener el vector original *x* a partir de un vector transformado *X* con la siguiente fórmula:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}; n = 0, \ldots, N-1 \quad (2)$$

Su complejidad temporal pertenece a $O(n^2)$ y la espacial a O(n).

FFT: Fast Fourier Transform

Este algoritmo produce la misma salida que el de DFT, con la diferencia que la obtiene eficientemente. Además, se ha desarrollado de manera que solo transforme vectores de longitud potencia de dos, en caso de ser menores se llena con ceros hasta la siguiente potencia de dos. Se basa en aplicar la técnica recursiva *dividir y conquistar* separando al vector de entrada *x* en dos subvectores *p* y *q* que contiene las componentes de índices pares e impares respectivamente. Utiliza la siguiente fórmula:

$$X(k) = P(k) + W_N^k Q(k); k = 0, \ldots, N-1 \quad (3)$$

En donde *P* y *Q* son las transformaciones *p* y *q* y el operador de acceso a estos arreglos trabaja en forma modular.

Su complejidad temporal pertenece a O(n log n) y la espacial a O(n).

IFFT: Inverse Fast Fourier Transform

Análogamente, este algoritmo produce la misma salida que IDFT pero de manera eficaz, utilizando *dividir y conquistar*. También fue hecho de tal manera que si recibe un vector que no tiene un largo potencia de dos es rellenado con ceros hasta tenerlo. Cumple la siguiente fórmula:

$$X(k) = \frac{1}{N}\left(P(k) + W_N^{-k} Q(k); k = 0, \ldots, N-1\right) \quad (4)$$

Nótese que las diferencias con respecto a FFT son el factor N^{-1} y el signo negativo en el exponente *k*. Su complejidad temporal pertenece a O(n log n) y la espacial a O(n).

Análisis de complejidad de los algoritmos

Análisis temporal de DFT e IDFT:

Analizando la función DFT queda que:

$$F(n) = O(1) + (O(n) * (O(n) * O(1) + O(1))$$

El primer O(1) proviene de la declaración y asignación de las variables *k*, *n*, *largo_in*, *vector_out* y *aux*. Luego en el código se puede encontrar un bloque *for* dentro de otro bloque *for* y dentro de este operaciones elementales además del *append*. Este bloque previamente presentado se representa como (O(n) * (O(n) * O(1) + O(1)), siendo n la cantidad de elementos en el vector pasado como argumento. Si bien el *append* es O(1) en nuestra implementación, algún cambio en su implementación podría cambiar considerablemente el tiempo de ejecución del algoritmo. Finalmente el tiempo de ejecución del algoritmo para el peor de los casos queda:

$$F(n) = O(1) + (O(n) * O(n))$$

$$F(n) = O(1) + O(n^2)$$

$$F(n) = O(n^2)$$

Para la función IDFT el análisis es el mismo, teniendo como resultado un tiempo de ejecución en el peor de los casos de $O(n^2)$.

Análisis espacial de DFT e IDFT:

Analizando la función DFT queda que:

$$F(n) = O(1) + O(n)$$

Correspondiendo O(1) a las variables *k* , *n* , *largo_in* y *aux*. En cambio la cantidad de memoria que se usa para *vector_out*, depende del largo del vector pasado como argumento, perteneciendo a O(n) (siendo n el largo del vector pasado como argumento). Finalmente el espacio de memoria que se usa es:

$$F(n) = O(n)$$

Para la función IDFT el análisis es el mismo, teniendo como resultado O(n).

Análisis temporal de FFT e IFFT:

Al observar la función *fft* se ve que se utilizan dos sub funciones llamadas *fill0till_exp2* y *_fft*.

La primera es la encargada de completar el vector con ceros hasta obtener un largo que sea potencia de dos. Esta función, al ser iterativa, es simple de analizar.

Se compone de una primera parte donde se hacen las declaraciones y algunos cálculos simples, los cuales son considerados de complejidad constante. Luego viene la parte iterativa donde se realizan "n + m" operaciones de asignación que toman un tiempo constante, siendo "n" la cantidad de complejos del vector original y "m" la cantidad necesaria para completar dicho vector hasta que su tamaño sea potencia de dos. Sabiendo que "m" toma como máximo un valor de n − 1, entonces podemos afirmar que la complejidad temporal de fill0till_exp2 es O(n).

La segunda función es la FFT estrictamente. Esta, al ser una función recursiva, es un poco más complicada de analizar que la anterior.

Se compone de una primera parte, de complejidad constante, donde se hacen las declaraciones, dos ciclos donde se realizan "n" operaciones constantes (en este caso "n" es la cantidad de complejos incluyendo los agregados por *fill0till_exp2*, pero se continuará utilizando esta variable ya que para la notación asintótica no hay diferencia), dos llamadas recursivas a la misma función pasando vectores de tamaño n/2 y por último otros dos ciclos donde entre ambos, se realizan otras "n" operaciones constantes. Por lo que la complejidad de *_fft* nos queda de la forma:

$$T\_fft(n) = O(1) + 2O(n) + 2T\_fft(n/2) + O(n) = O(n) + 2T(n/2)$$

Una vez llegado a este punto, se puede utilizar el método de expansión, obteniendo así la complejidad temporal de *_fft* que resulta en:

$$T\_fft(n) = O(nlog(n))$$

Por último al tener la complejidad de ambas sub funciones se las puede sumar para obtener la complejidad de *fft*:

$$T(n) = O(n) + O(nlog(n)) = O(nlog(n))$$

La función *ifft* al tener el mismo formato que la *fft* puede ser analizada exactamente de la misma manera, llegando al mismo resultado.

Análisis espacial de FFT e IFFT:

Para poder analizar la complejidad espacial de *fft* vamos a analizar primero las funciones que la componen.

Observando *fill0till_exp2* se nota que la memoria utilizada es lineal con la cantidad de complejos del vector ya que el resto de la memoria pedida es constante y por lo tanto es despreciada por la notación asintótica.

Posteriormente, se ve que *_fft* posee una primera parte en la asignación de variables donde se hace un pedido de memoria que puede considerarse constante, luego se crean dos vectores de largo n /2 y se realizan las llamadas recursivas, en el resto del código no se ejecutan pedidos de memoria. Por lo tanto, la complejidad espacial queda de la forma:

$$E\_fft(n) = O(1) + 2O(n/2) + 2E\_fft(n/2) = O(n) + 2E\_fft(n/2) = O(nlog(n))$$

Por último al sumar el coste temporal de ambas funciones obtenemos que la complejidad temporal de *fft* es de:

$$E(n)) = O(n) + O(nlog(n)) = O(nlog(n))$$

Por la misma razón que se ha expuesto antes, la función *ifft* posee la misma complejidad espacial que la fft.

Análisis temporal del main:

En el main, si no se considera el costo de las transformadas de Fourier, el único momento en que se realiza una operación no constante es cuando se lee la línea de complejos del archivo, y este proceso tiene un coste lineal con la cantidad de complejos presentes en dicha línea. Luego al observar el coste temporal de las diferentes transformadas de Fourier vemos que todas superan ampliamente este coste. Por lo tanto, podemos afirmar que el coste temporal del programa está directamente ligado al de la transformada utilizada.

Análisis espacial del main:

Este caso es diferente al anterior ya que para lograr que el coste temporal del programa sea esencialmente el de la transformada utilizada se optó por utilizar memoria de más. Por lo tanto, al analizar la memoria utilizada, podemos notar que esta se magnifica al leer la línea del archivo, ya que se reserva espacio para tantos complejos como la mitad más uno de los caracteres de dicha línea. De esta forma se logra que en ningún punto del programa sea necesario aumentar la capacidad de los diferentes vectores aliviando significativamente el coste temporal del mismo.

En el resto del programa la memoria utilizada es a lo sumo O(nlog(n)) siendo n la cantidad de complejos de una línea del archivo pero en el peor caso (y en la gran mayoría) este valor es ínfimo comparado la cantidad de caracteres que posee dicha línea. Por lo tanto, afirmamos que la complejidad espacial del programa es, siendo c la cantidad de caracteres de cada línea.:

$$E(c) = O(c/2 + 1) = O(c)$$

Proceso de compilación

Para la obtención del ejecutable se optó por hacer un *Makefile*, en el cual también se pueden realizar las pruebas con los argumentos correspondientes.

Entre las posibles pruebas que se pueden hacer están la del correcto funcionamiento de la clase vector_t, la comprobación de que no haya pérdidas de memoria y la de las funciones DFT, IDFT, FFT e IFFT.

Comandos para makefile

make programa: Genera el ejecutable del programa.

test_programa_dft: Prueba el programa utilizando archivos de prueba que tienen el formato test_ft<número>, test_ift<número>, test_fft<número> o test_ifft<número>. Estos archivos de prueba se pasan automáticamente a test_diff donde se compara los archivos de prueba y referencia para verificar si hay un error en el programa.

test_programa_valgrind: Prueba el programa utilizando archivos de pruebas con el mismo formato que test_programa_dft y comprueba que no haya pérdida de memoria con *valgrind* –leack-check= full.

test-vector_t: Prueba la clase vector_t con el código test_vector_t.o. Se usa un archivo de prueba test_vector_t.txt. El archivo está compuesto por igualdades de este tipo : <entrada> = <resultado esperado> y por "comentarios" que especifican al programa que methodo probar. Las opciones son:

- #STREAM: Prueba operador >> y << del vector_t.
- #LARGO: Prueba el largo del vector_t y como se actualiza.
- #== : Prueba el operador == en vector_t.

Además se utilizan los métodos append() y clean() en cada una de las opciones, para probar los métodos mencionados.

test-vector_t-valgrind: Prueba el vector_t como en el comando anterior, utilizando el mismo formato de archivo de prueba. Además usa valgrind para comprobar si hubo pérdidas de memoria. *Valgrind* se usa con el flag –leak-check=full.