LENGUAJES DE PROGRAMACIÓN

Trabajo Práctico - Septiembre de 2020

Alumno: Javier García Parra

Índice

1. Introducción

Esta es la memoria del trabajo práctico realizado por el alumno Javier García Parra para la asignatura de lenguajes de programación en la convocatoria de Septiembre de 2020.

Para facilitar la depuración y su ejecución, todos los ejercicios comparten el mismo punto de entrada (solo hay un método main en la aplicación).

La estructura de la aplicación es:

- main.cpp: punto de entrada de la aplicación.

- ejercicio1.h ejercicio1.cpp: la funcion principal es ejercicio1, el resto son auxiliares.

- ejercicio2.h ejercicio2.cpp: la funcion principal es ejercicio2, el resto son auxiliares.

- ejercicio3h ejercicio3.cpp: la funcion principal es ejercicio3, el resto son auxiliares.

- ejercicio4.h ejercicio4.cpp: la funcion principal es ejercicio4 el resto son auxiliares.

- utils.h utils.cpp: funciones auxiliares compartidas en los diferentes ejercicios.

- files/: directorio donde se almacenan los archivos necesarios para la ejecución normal y el testado de los ejercicios.

- CmakeLists.txt: fichero de parametrización de cmake.

2. Compilación y ejecución

Dado que el desarrollo ha sido con el ide Clion y este emplea Cmake por defecto, se ha mantenido este set de herramientas para su compilación.

Se proporcionan a continuación las instrucciones para la compilación en linux (s.o. en el que se ha desarrollado), pero cmake es una herramienta multiplataforma, así que en otro s.o. el proceso será similar. También se proporciona el comando para su compilación con g++.

2.1. Con cmake

Para su compilación con cmake y su ejecución emplear los siguientes comandos:

```
$ cmake -Bbuild -H.

$ cmake --build build --target all

$ build/./septiembre
```

2.2. Con g++

Para su compilación con g++ y su ejecución emplear los siguientes comandos:

```
$ g++ main.cpp ejercicio1.cpp ejercicio2.cpp ejercicio3.cpp ejercicio4.cpp utils.cpp

$ ./a.out
```

3. Inicio

Al arrancar el programa, este pregunta qué ejercicio se desea ejecutar. El 0 sirve para ejecutarlos todos, los restantes números del 1 al 4 corresponden a cada uno de los ejercicios empleados.

Una vez seleccionado el ejercicio se pregunta si se desea ejecutar en modo depuración. Si se escribe ‘s’ (de sí), en la ejecución de cada ejercicio no se darán opciones de ejecución al usuario (tomando valores por defecto) harán una serie de comprobaciones en la ejecución, mostrando errores a lo largo de la ejecución de los tests, si los hay, y mostrando al final si el resultado final es correcto o no.

La forma más rápida de ver la ejecución es:

```
Que ejercicio desea iniciar? (0-4, 0 para todos) 0

Modo pruebas? (s/n) s
```

Con las opciones anteriores se ejecutarán todos los ejercicios en modo pruebas, con lo que se cargarań las opciones por defecto y evaluarán los resultados.

3.1. Código

El código de main.cpp se recoge a continuación. Su función es ejecutar el/los ejercicios según las opciones seleccionadas por el usuario.

```
#include "ejercicio1.h"
#include "ejercicio2.h"
#include "ejercicio3.h"
#include "ejercicio4.h"
#include "utils.h"
int main() {
	int ejercicio;
	bool debug;
	ejercicio = askForInt("Que ejercicio desea iniciar? (0-4, 0 para todos) ");
	debug = askForChar("Modo pruebas? (s/n) ") == 's';
	switch (ejercicio) {
		case 0:
			ejercicio1(debug);
			ejercicio2(debug);
			ejercicio3(debug);
			ejercicio4(debug);
			return 0;
		case 1:
			return ejercicio1(debug);
		case 2:
			return ejercicio2(debug);
		case 3:
			return ejercicio3(debug);
		case 4:
			return ejercicio4(debug);
		default:
```

```
                printError("no existe!");
        }
        return 0;
}
```

4. Ejercicio 1

Para la implementación de la evaluación del polinomio de forma eficaz se ha empleado el algoritmo de Horner ([1] p. 77).

4.1. Salida en modo normal

En modo normal el usuario debe introducir el orden del polinomio y los coeficientes del mismo. Estos últimos deben ir seguidos de Enter después de introducir el número y se pedirán n+1.

Una vez introducidos, se mostrará el resultado de la evaluación de los N valores calculados según la fórmula descrita en el enunciado.

```
***************
* EJERCICIO 1 *
***************
Evaluacion de polinomios


¿Orden del polinomio? 3
Introduce los coeficientes:


1
2
3
4
Evaluacion del polinomio para los 10 valores de x_k = cos(2*pi*k/N)k: 0 pn(x_k)= 4.00000000e+00
k: 1 pn(x_k)= 3.80901699e+00
k: 2 pn(x_k)= 3.30901699e+00
k: 3 pn(x_k)= 2.69098301e+00
k: 4 pn(x_k)= 2.19098301e+00
k: 5 pn(x_k)= 2.00000000e+00
k: 6 pn(x_k)= 2.19098301e+00
k: 7 pn(x_k)= 2.69098301e+00
```

```
k: 8 pn(x_k)= 3.30901699e+00
k: 9 pn(x_k)= 3.80901699e+00
```

4.2. Salida en modo depuración

En modo de depuración se asume un orden de polinomio 1 con coeficientes 1.

Para la comprobación de los resultados se comparan los resultados de k = 1 y k = 2 con los esperados.

```
***************
* EJERCICIO 1 *
***************
Evaluacion de polinomios


Introduce los coeficientes:


Evaluacion del polinomio para los 10 valores de x_k = cos(2*pi*k/N)k: 0 pn(x_k)= 2.00000000e+00
resultado correcto
k: 1 pn(x_k)= 1.80901699e+00
resultado correcto
k: 2 pn(x_k)= 1.30901699e+00
resultado correcto
k: 3 pn(x_k)= 6.90983006e-01
k: 4 pn(x_k)= 1.90983006e-01
k: 5 pn(x_k)= 0.00000000e+00
k: 6 pn(x_k)= 1.90983006e-01
k: 7 pn(x_k)= 6.90983006e-01
k: 8 pn(x_k)= 1.30901699e+00
k: 9 pn(x_k)= 1.80901699e+00
```

4.3. Código

Al igual que los restantes ejercicios, existe una función principal (llamada desde main.cpp).

La única función auxiliar es la que se encarga de comprobar los resultados.

```
#include <iostream>
#include <vector>
```

```
#include <cmath>
#include "ejercicio1.h"
#include "utils.h"
using namespace std;
const int n_debug = 1;
const int N = 10;
const double pi = 3.14159265358979323846;
/*
 * Comprobacion de resultados, se comparan frente a los esperados
 */
void checkResult(unsigned int k, double valor) {
	bool resultOk = true;
	switch (k) {
		case 0:
			if (!areEqual(valor, 2)) resultOk = false;
			printResult(resultOk);
			break;
		case 1:
			if (!areEqual(valor, 1.80901699)) resultOk = false;
			printResult(resultOk);
			break;
		case 2:
			if (!areEqual(valor, 1.30901699)) resultOk = false;
			printResult(resultOk);
			break;
	}
}
/*
 * Funcion principal del ejercicio 1.
 * - Pregunta por el orden del polinomio. Si es modo debug, se carga el definido en n_debug
 * - Pregunta por los coeficientes. Si es modo debug, todos los coef. son iguales a 1.
 * - Muestra los valores calculados de evaluar el polinomio, siguiendo el algoritmo de Horner,
para los
 * valores de x_k calculados con x_k = cos( 2 * pi * k / N)
 */
int ejercicio1(bool debug) {
	printBanner("EJERCICIO 1");
	print("Evaluacion de polinomios");
	// Introduccion del orden del polinomio
	int n = debug ? n_debug : askForInt("¿Orden del polinomio? ");
	if (n < 0) {
		printError("No puede ser negativo");
		return 1;
	}
	// Introduccion de coeficientes
	print("Introduce los coeficientes:");
	vector<double> coefs(n + 1, 1);
	if (!debug) {
		double coef;
		for (unsigned int i = 0; i < n + 1; i++) {
			cin >> coef;
			coefs.push_back(coef);
		}
	}
	// Evaluación del polinomio para los N x_k valores.
	cout << "Evaluacion del polinomio para los " << N << " valores de x_k = cos(2*pi*k/N)";
```

```
        setPrecision(8);
        setScientific();
        for (unsigned int k = 0; k < N; k++) {
                // Implementación del algoritmo de Horner
                // p(x) = q1(q2(..(qn(x))..)
                double x = cos(2 * pi * k / N);
                for (unsigned int j = n; j > 1; j--) {
                        x = coefs.at(j - 1) + coefs.at(j) * x;
                }
                double valor = coefs.at(0) + x;
                cout << "k: " << k << " pn(x_k)= " << valor << endl;
                if (debug) {
                        checkResult(k, valor);
                }
        }
        return 0;
}
```

4.4. Testado

Para testar se asume un polinomio

p(x) = 1 + x

y se comparan los resultados calculados con los esperados para los dos primeros valores de k.

5. Ejercicio 2

5.1. Salida en modo normal

El usuario debe seleccionar un punto de inicio. Además de los requeridos se ha añadido un tercero con t muy superior al de los anteriores, para comprobar que se alcanzaba el contorno. Con los otros dos puntos de inicio normalmente se alcanza el punto inicial.

```
***************
* EJERCICIO 2 *
***************
Estimacion mediante metodo Monte Carlo


Selecciona un punto de inicio:
 1 - (0.4, 0.04)
 2 - (0.4, 0.1)
 3 - (0.4, 100) 2
Recorriendo 10000 caminos...
```

```
(x*,t*)=(0.4,0.04) -> U(x*,t*) = 100


0 alcanzaron el contorno, 10000 alcanzaron el inicio.
```

5.2. Salida en modo depuración

En modo depuración se asume el punto (0.4, 0.04) como inicio.

```
***************
* EJERCICIO 2 *
***************
Estimacion mediante metodo Monte Carlo


Recorriendo 10000 caminos...


(x*,t*)=(0.4,0.04) -> U(x*,t*) = 100


0 alcanzaron el contorno, 10000 alcanzaron el inicio.


resultado correcto
```

5.3. Código

```
#include <iostream>
#include <vector>
#include <cmath>
#include <numeric>
#include "ejercicio1.h"
#include "utils.h"
using namespace std;
const int l = 200;
const int Nc = 10000;
/*
 * Funcion principal del ejercicio2.
 * - Estimacion de una magnitud fisica en el punto (x*,t*) en base al metodo de Monte Carlo.
 */
int ejercicio2(bool debug) {
	printBanner("EJERCICIO 2");
	print("Estimacion mediante metodo Monte Carlo");
	// Seleccion del punto (x*, t*)
	// El usuario seleccionara el punto de partida. Se ha incluido un tercer punto con t mucho
mayor a los del enunciado
	// para comprobar que el algoritmo alcanza el contorno.
	double x_ast = 0.4;
```

```
	double t_ast = 0.04;
	if (!debug) {
		int puntoInicio = askForInt(
				"Selecciona un punto de inicio: \r 1 - (0.4, 0.04) \r 2 - (0.4, 0.1) \r 3 - (0.4,
100) ");
		if (puntoInicio == 2) {
			t_ast = 0.1;
		} else if (puntoInicio == 3) {
			t_ast = 100;
		}
	}
	// Discretizacion de las variables x y t
	double delta_x = (double) 1 / l;
	double delta_t = (double) 1 / (2 * l ^ 2);
	// Recorrido de los Nc caminos, a partir de (x*, t*), almacenando los resultados de cada final
	// en el vector Ubs segun:
	// - si se alcanza t = 0, U(x,0) = 100
	// - si se alcanza el contorno (x=0 o x=1), U(0,t) = 0 = U(1,t)
	cout << "Recorriendo " << Nc << " caminos..." << endl << endl;
	int k = 1, nContorno = 0, nInicio=0;
	double x, t;
	vector<double> Ubs;
	double Ub;
	do {
		x = x_ast;
		t = t_ast;
		do {
			// Generacion de un numero pseudoaleatorio
			double r = ((double) random() / (RAND_MAX));
			// Desplazamiento en base a r
			if (r < 0.25) {
				x += delta_x;
			} else if (r < 0.5) {
				x -= delta_x;
			} else {
				t -= delta_t;
			}
		} while (t > 0 && x > 0 && x < 1);
		if(t>0){
			Ub = 0;
			nContorno ++;
		}else{
			Ub = 100;
			nInicio++;
		}
		Ubs.push_back(Ub);
		k++;
	} while (k <= Nc);
	// Calculo del promedio de los valores almacenados en el vector Ub
	double result = std::accumulate(Ubs.begin(), Ubs.end(), 0) / Nc;
	setPrecision(3);
	cout << "(x*,t*)=(" << x_ast << "," << t_ast << ") -> " << "U(x*,t*) = " << result <<
endl << endl;
	cout << nContorno << " alcanzaron el contorno, " << nInicio << " alcanzaron el inicio." <<
endl;
```

```
	if (debug) {
		bool resultOk = Ubs.size() == Nc && ( nContorno + nInicio ) == Nc;
		printResult(resultOk);
	}
	return 0;
}
```

5.4. Testado

Dado que el resultado depende de una variable aleatoria, se comprueba que el tamaño del vector Ub coincide con el número de caminos establecido y que el número de soluciones en que se alcanza el contorno y el del número de veces que se alcanza el contorno suman el mismo número de caminos.

6. Ejercicio 3

Para la implementación del método de diferencias divididas se ha seguido la definición del método de Newton del libro [1] p125.

6.1. Salida en modo normal

Se pregunta al usuario qué fichero de puntos se desea cargar (de los ficheros definidos en el enunciado).

```
***************
* EJERCICIO 3 *
***************
Interpolacion mediante el metodo de Newton de las diferencias divididas


¿Que fichero de puntos desea cargar?
 1. Primero
 2. Segundo
1
Cargando fichero de puntos files/puntos1.txt
Tabla de diferencias divididas:


0	-5	6	2	1
1	1	12	6
3	25	30
```

```
4      55


Evaluando polinomio de interpolacion para x = 0.5


Resultado de la evaluacion:


f(0.5) = -1.875
```

6.2. Salida en modo depuración

En este modo se carga un fichero con los datos del ejercicio descrito en [1] p.128. El resultado debe ser igual al del resultado del ejercicio.

```
***************
* EJERCICIO 3 *
***************
Interpolacion mediante el metodo de Newton de las diferencias divididas


Ejecutando tests...


Cargando fichero de puntos files/fake.txt
Cargando fichero de puntos files/puntos_dup.txt
Cargando fichero de puntos files/puntos_single.txt
Cargando fichero de puntos files/puntos1.txt
Todos los test pasados con éxito.


Cargando fichero de puntos files/puntos_test.txt
Tabla de diferencias divididas:


0.1   0.7   1     5     -25   62.5 -125
0.2   0.8   2     -2.5  6.3652787e-14 -2.9235873e-13
0.3   1     1.5   -2.5  -5.3290705e-14
0.4   1.15  1     -2.5
0.5   1.25  0.5
0.6   1.3
```

```
Evaluando polinomio de interpolacion para x = 0.55

0.7+1 * (x - 0.1) +5 * (x - 0.1) (x - 0.2) -25 * (x - 0.1) (x - 0.2) (x - 0.3) +63 * (x - 0.1) (x - 0.2) (x - 0.3) (x - 0.4) -
1.3e+02 * (x - 0.1) (x - 0.2) (x - 0.3) (x - 0.4) (x - 0.5)


Resultado de la evaluacion:


f(0.55) = 1.2853516


resultado correcto

Process finished with exit code 0
```

6.3. Código

La función principal es ejercicio3, las restantes son funciones auxiliares cuya finalidad se describe sobre la definición de cada función.

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <fstream>
#include <algorithm>
#include <iomanip>
#include "ejercicio1.h"
#include "utils.h"
using namespace std;
string FILE_PATH_1 = "files/puntos1.txt";
string FILE_PATH_2 = "files/puntos2.txt";
string FILE_PATH_DEBUG = "files/puntos_test.txt";
bool DEBUG = false;
int DEBUG_ERROR = 0;
struct Punto {
	double x;
	double fx;
};
/*
 * Comprueba si el punto ya existe en el vector
 */
bool puntoExist(Punto punto, std::vector<Punto> puntos) {
	return any_of(puntos.begin(), puntos.end(), [punto](Punto p) {
		return p.x == punto.x && p.fx == punto.fx;
	});
}
/*
 * Carga del fichero de puntos.
 */
std::vector<Punto> readFile(string filePath) {
	cout << "Cargando fichero de puntos " << filePath << endl;
	std::ifstream ifs(filePath);
```

```
	std::string buffer;
	std::vector<Punto> puntos;
	if (!ifs) {
		if (!DEBUG) {
			cerr << "Error abriendo archivo: " << filePath.c_str();
			exit(EXIT_FAILURE);
		} else {
			DEBUG_ERROR = 1;
			return puntos;
		}
	}
	std::string line;
	while (getline(ifs, line)) {
		std::stringstream ss(line);
		std::vector<std::string> item;
		std::string tmp;
		Punto punto;
		while (getline(ss, tmp, ' ')) {
			item.push_back(tmp);
		}
		punto.x = atof(item.at(0).c_str());
		punto.fx = atof(item.at(1).c_str());
		// Comprueba duplicados
		if (!puntoExist(punto, puntos)) {
			puntos.push_back(punto);
		} else {
			if (!DEBUG) {
				cerr << "Se ha encontrado un punto duplicado (" << punto.x << "," <<
punto.fx << ")";
				exit(EXIT_FAILURE);
			} else {
				DEBUG_ERROR = 2;
				return puntos;
			}
		}
	}
	// Comprueba si hay menos de dos puntos
	if (puntos.size() < 2) {
		if (!DEBUG) {
			printError("El fichero debe tener menos de 2 puntos.");
			exit(EXIT_FAILURE);
		} else {
			DEBUG_ERROR = 3;
			return puntos;
		}
	}
	return puntos;
}
/*
 * Ejecuta una batería de pruebas para asegurar que se cumplen los requisitos
 */
void runTests() {
	print("Ejecutando tests...");
	bool allPasssed = true;
	// Si hay un error al cargar el fichero
```

```
	readFile("files/fake.txt");
	if (DEBUG_ERROR != 1) {
		printError("Primer test ha fallado.");
		allPasssed = false;
	}
	// Si hay duplicados
	DEBUG_ERROR = 0;
	readFile("files/puntos_dup.txt");
	if (DEBUG_ERROR != 2) {
		printError("Segundo test ha fallado.");
		allPasssed = false;
	}
	// Si hay un solo punto
	DEBUG_ERROR = 0;
	readFile("files/puntos_single.txt");
	if (DEBUG_ERROR != 3) {
		printError("Tercer test ha fallado.");
		allPasssed = false;
	}
	// Si no hay errores en la carga
	DEBUG_ERROR = 0;
	std::vector<Punto> puntos = readFile("files/puntos1.txt");
	if (DEBUG_ERROR != 0 | puntos.at(0).x != 0 | puntos.at(0).fx != -5) {
		printError("Cuarto test ha fallado.");
		allPasssed = false;
	}
	if (allPasssed) {
		print("Todos los test pasados con éxito.");
	}
}
/*
 * Funcion principal del ejercicio 3.
 * - Interpolacion mediante el metodo de Newton de las diferencias divididas.
 */
int ejercicio3(bool debug) {
	printBanner("EJERCICIO 3");
	print("Interpolacion mediante el metodo de Newton de las diferencias divididas");
	// se almacena en una variable global
	DEBUG = debug;
	double x;
	std::vector<Punto> puntos;
	int n;
	// Determinacion del fichero de entrada de puntos
	// Para la depuracion se cargan datos extraidos del libro 'Introducion al calculo numerico'
Carlos Moreno p.128
	// En modo normal se pregunta al usuario cual de los ficheros de datos del enunciado desea
cargar.
	// Al finalizar la carga se ordenan los datos.
	string filePath;
	if (DEBUG) {
		runTests();
		filePath = FILE_PATH_DEBUG;
	} else {
		filePath = askForInt("¿Que fichero de puntos desea cargar?\n 1. Primero\n 2. Segundo\
n") == 1 ?
```

```
                        FILE_PATH_1 : FILE_PATH_2;
    }
    puntos = readFile(filePath);
    sort(puntos.begin(), puntos.end(), [](Punto &p1, Punto &p2) { return p1.x < p2.x; });
    // Se carga el valor de x del punto que se desea interpolar y se comprueba que pertenece al dominio.
    x = DEBUG ? 0.55 : 0.5;
    if (x < puntos.at(0).x | x > puntos.at(puntos.size() - 1).x) {
        printError("El valor no está en el dominio de los puntos definidos en fichero.");
        exit(EXIT_FAILURE);
    }
    // Orden del polinomio de interpolacion
    n = puntos.size() - 1;
    // Implementacion del metodo de diferencias divididas
    double diff_div[n + 1][n + 2];
    // - relleno las dos primeras columnas
    for (int i = 0; i <= n; i++) {
        diff_div[i][0] = puntos.at(i).x;
        diff_div[i][1] = puntos.at(i).fx;
    }
    // - relleno la tabla de diferencias divididas
    for (int j = 2; j <= n + 1; j++) {
        for (int i = 0; i <= n; i++) {
            diff_div[i][j] = (diff_div[i + 1][j - 1] - diff_div[i][j - 1]) / (diff_div[i + j - 1][0] - diff_div[i][0]);
        }
    }
    // Muestra la tabla de diferencias divididas
    print("Tabla de diferencias divididas:");
    setPrecision(3);
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n + 1; j++) {
            if (j < 2 | j < n - i + 2)
                cout << diff_div[i][j] << "\t";
        }
        cout << "\n";
    }
    cout << endl;
    // Evaluacion del polinomio de interpolacion.
    // En modo pruebas muestra el polinomio
    cout << "Evaluando polinomio de interpolacion para x = " << x;
    double interpolatedValue = diff_div[0][1];
    if (DEBUG) cout << "\n" << interpolatedValue;
    for (int j = 2; j <= n + 1; j++) {
        double productTerm = 1;
        if (DEBUG) {
            if (diff_div[0][j] > 0) cout << "+";
            cout << diff_div[0][j] << " * ";
        }
        for (int i = 0; i < j - 1; i++) {
            if (DEBUG) cout << setprecision(2) << "(x - " << diff_div[i][0] << ") ";
            productTerm = productTerm * (x - diff_div[i][0]);
        }
        interpolatedValue = interpolatedValue + (productTerm * diff_div[0][j]);
    }
```

```
	cout << endl << endl;
	// Muestra el valor interpolado
	print("Resultado de la evaluacion:");
	setPrecision(3);
	cout << "f(" << x << ") = " << interpolatedValue << endl;
	if (DEBUG) {
		printResult(areEqual(interpolatedValue, 1.28535156));
	}
	return 0;
}
```

6.4. Testado

Para testar las funcionalidades requeridas se ejecutan una serie de tests dentro de la función runTest(). Si alguna de las funcionalidades no se cumple se muestra

Para testar el cálculo final se carga un fichero con los datos del ejercicio descrito en [1] p.128. El resultado debe ser igual al del resultado del ejercicio.

7. Ejercicio 4

Para la implementación se emplea un vector para los nombres y una clase llamada Grafo para la información del grafo, empleando listas de adyacencia de cada uno de los vértices (almacenando cada vértice como un entero que se asocia con su nombre en el vector vertices). La solución está basada en [2].

7.1. Salida en modo normal

No es necesaria la interacción por parte del usuario. Si se desea cargar otro grafo, se deberán modificar directamente los ficheros files/vertices.txt y files/arcos.txt.

```
***************
* EJERCICIO 4 *
***************
Representacion de grafo no dirigido.


1e4 :
2k1 : 3
3 : 2k1 3Ab
3Ab : 3
A : e4 A A
```

```
E4 : E4 E4
aR12 : e3
e1 : e2 e1 e1 e3
e2 : e1 e3
e3 : e1 e4 e2 aR12
e4 : e3 A
```

7.2. Salida en modo depuración

La salida difiere del normal en la última linea, en la que se muestra si el resultado es correcto o no.

```
***************
* EJERCICIO 4 *
***************
Representacion de grafo no dirigido.


1e4 :
2k1 : 3
3 : 2k1 3Ab
3Ab : 3
A : e4 A A
E4 : E4 E4
aR12 : e3
e1 : e2 e1 e1 e3
e2 : e1 e3
e3 : e1 e4 e2 aR12
e4 : e3 A


resultado correcto
```

7.3. Código

La función principal es ejercicio4, el resto son funciones auxiliares.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <sstream>
#include <algorithm>
```

```
#include "ejercicio1.h"
#include "utils.h"
using namespace std;
string FILE_PATH_EDGES = "files/arcos.txt";
string FILE_PATH_VERTICES = "files/vertices.txt";
// Estructura del arco que une los vertices
struct Arco {
	int src, dest;
};
// Vectores para almacenar vertices y arcos
vector<string> vertices;
vector<Arco> arcos;
// Implementacion del grafo, mediante listas de adyacencia de enteros.
class Grafo {
public:
	vector<vector<int>> adjList;
	// Constructor para vector de arcos y numero de vertices
	Grafo(vector<Arco> const &arcos, int N) {
		adjList.resize(N);
		// Asociacion de vertices en ambas direcciones
		for (auto &arco: arcos) {
			adjList[arco.src].push_back(arco.dest);
			adjList[arco.dest].push_back(arco.src);
		}
	}
};
/*
 * Muestra el grafo
 */
void printGraph(Grafo const &grafo, int N) {
	for (int i = 0; i < N; i++) {
		cout << vertices.at(i) << " : ";
		for (int v : grafo.adjList[i])
			cout << vertices.at(v) << " ";
		cout << endl;
	}
}
/*
 * Carga del fichero de vertices.
 * Los nombres de los vertices se almacenan en un vector que permite asociar
 * la cadena del nombre con el entero que representa al vertice en el grafo.
 */
vector<string> readVertices() {
	std::ifstream ifs(FILE_PATH_VERTICES);
	std::string buffer;
	vector<string> vertices;
	if (!ifs) {
		cerr << "Error abriendo archivo: " << FILE_PATH_VERTICES.c_str();
		exit(EXIT_FAILURE);
	}
	std::string line;
	while (getline(ifs, line)) {
		vertices.push_back(line);
	}
	sort(vertices.begin(), vertices.end());
	return vertices;
```

```
}
/*
 * Devuelve el entero asociado al nombre del vertice.
 */
int getIndexOfVertice(string vertice) {
	vector<string>::iterator it = find(vertices.begin(), vertices.end(), vertice);
	int pos = distance(vertices.begin(), it);
	return pos;
}
/*
 * Carga del fichero de arcos.
 */
vector<Arco> readArcos() {
	std::ifstream ifs(FILE_PATH_EDGES);
	std::string buffer;
	if (!ifs) {
		cerr << "Error abriendo archivo: " << FILE_PATH_EDGES.c_str();
		exit(EXIT_FAILURE);
	}
	std::string line;
	while (getline(ifs, line)) {
		std::stringstream ss(line);
		std::vector<std::string> item;
		std::string tmp;
		//Punto punto;
		while (getline(ss, tmp, ' ')) {
			item.push_back(tmp);
		}
		Arco edge = {
				getIndexOfVertice(item.at(0)),
				getIndexOfVertice(item.at(1))
		};
		arcos.push_back(edge);
	}
	return arcos;
}
/*
 * Funcion principal del ejercicio 4.
 * - Lee los ficheros de vertices y arcos.
 * - Almacena la informacion del grafo.
 * - Muestra la informacion del grafo.
 */
int ejercicio4(bool debug) {
	printBanner("EJERCICIO 4");
	print("Representacion de grafo no dirigido.");
	// Carga de ficheros
	vertices = readVertices();
	arcos = readArcos();
	int N = vertices.size();
	// Construccion del grafo
	Grafo graph(arcos, N);
	// Muestra el grafo
	printGraph(graph, N);
	// Comprobacion de que la importacion ha sido correcta
	if (debug) {
```

```
            bool resultOk = graph.adjList[0].size() == 0 && graph.adjList[1].size() == 1 &&
graph.adjList[2].size() == 2;
            printResult(resultOk);
      }
      return 0;
}
```

7.4. Testado

El testado se basa en la comprobación del tamaño de las listas de adyacencia de los vertices 1e4, 2k1 y 3.

8. Referencias

[1] Introducción al análisis numérico. Carlos Moreno González. Uned. 2011.

[2] https://www.techiedelight.com/graph-implementation-using-stl/