PRACTICA AVANCE RÁPIDO Y BACKTRACKING

AÑO ACADÉMICO: 2020-2021

ASIGNATURA: AED2

ALUMNOS: David Fernández Expósito y

Pablo Tadeo Romero Orlowska

E-MAIL: [redacted]

DNI: [redacted]

GRUPO Y SUBGRUPO: PCEO

PROFESOR: NORBERTO MARÍN PÉREZ

CUENTA MOOSHAK: PCEO_09

PROBLEMAS: F (BA), I (AR)

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

En este trabajo se utilizarán las técnicas de avance rápido y backtracking para resolver los problemas asignados, en este caso el I de avance rápido y el F de backtracking.

En cuanto a los archivos entregados en el .rar, se puede encontrar esta memoria en pdf, así como los archivos de código. El archivo llamado "I AR.cpp" es el código del problema de avance rápido, y el archivo "F BT.cpp" es el código del problema de backtracking. También se incluyen varios archivos Excel donde se podrán encontrar las tablas que dan lugar a las gráficas mostradas en este documento, además de un par de generadores y un pdf con algún ejemplo significativo.

2. LISTADO DE ENVÍOS

2.1 ALGORITMO VORAZ

Hicimos los envíos: 47 (Wrong answer), 48 (accepted), 120 (accepted) y 242 (accepted).

Los dos primeros envíos, aunque el segundo fuese aceptado, no seguían de manera muy estricta el esquema de avance rápido. La función seleccionar no quedaba muy clara, y ese fue principalmente el problema que provocó que no se aceptase el ejercicio en el primer envío. Luego conseguimos solucionarlo, pero aún seguía siendo poco claro el código. Por ese motivo se hizo el tercer envío, que ya sigue el esquema de manera más clara y la función seleccionar es más correcta. No obstante, nos dimos cuenta de que podíamos evitar un bucle que usábamos para calcular el beneficio final calculando este beneficio conforme se van añadiendo alumnos a la solución, por lo que decidimos hacer un cuarto y último envío para mejorar este aspecto.

2.2 BACKTRACKING

Hicimos los envíos: 26 (Compile Time Error), 27 (Time Limit Exceeded), 32 (Wrong answer), 33 (Wrong answer), 38 (Wrong answer), 44 (Wrong answer), 45 (Wrong answer), 47 (Wrong answer), 65 (Wrong answer), 123 (Accepted), 232 (Accepted), 322 (Accepted), 325 (Accepted) y 467 (Accepted).

El primer envío dio error de compilación simplemente porque declarábamos una variable que finalmente no usamos (era una variable para almacenar la solución como tal, pero el problema no nos pide eso, nos pide solo el precio máximo). El segundo envío era sumamente ineficiente pues recorríamos todo el árbol, no comprobábamos si no hacía falta continuar porque se había superado el presupuesto o si la solución óptima actual ya era la máxima posible. El resto de

envíos, todos con Wrong Answer, comparten el mismo problema. No seguíamos de forma correcta el esquema de backtracking y pusimos algún criterio que no era adecuado. Fuimos cambiando cosas pero no solucionábamos el problema, de ahí todos los envíos erróneos. Cuando pudimos tratar el problema con el profesor, Norberto, seguimos sus indicaciones y finalmente el mooshak nos lo aceptó. No obstante, en este envío no se hacen podas (o casi ninguna), por lo que hicimos un siguiente envío donde se implementó una poda muy similar a la explicada en clase para el problema de la asignación de tareas. Pero este envío se mejoró, pues tras hablar con el profesor, Norberto, nos dimos cuenta de que se podía hacer otra poda más, por lo que se introdujo esa poda y se volvió a enviar. Sin embargo, vimos que se podía mejorar el tiempo cuando nos encontrásemos con un mejor caso y decidimos hacer un penúltimo envío añadiendo esta mejora. El último envío simplemente solucionaba un pequeño error a la hora de inicializar las matrices, que no afectaba a nada porque luego se sobrescribía el valor con lo que se lee por pantalla.

3. AVANCE RÁPIDO

Como ya se ha comentado, nos tocó hacer el problema I de avance rápido.

3.1 PSEUDOCÓDIGO

3.1.1 STRUCT PAREJA

```
Tipo pareja{
        x, y: enteros;
}
```

Se decidió usar un tipo “pareja”, pues en el algoritmo diseñado se busca por parejas, y se va seleccionando por parejas, por lo que de esta manera podemos devolver los dos valores sin problema.

3.1.2 ALGORITMO VORAZ

En el problema se nos proporcionan 2 matrices con las que se calcula el “beneficio” que supone sentar un alumno con otro. Claramente, sentar a un alumno solo no supone ningún beneficio. Por lo tanto, para que fuese todo más sencillo se decidió trabajar con una matriz “sumas”, que se obtiene a partir de las otras 2, y que almacena el beneficio de sentar un alumno con otro. Nótese que, a pesar de que las dos matrices que se proporcionan no tienen por qué ser simétricas, esta

matriz que se calcula sí que lo es, lo que permite recorrerla sólo por la mitad superior, por ejemplo.

La idea que usamos para nuestro algoritmo voraz fue ir cogiendo parejas hasta que no queden alumnos o se quede uno suelto, en ese caso sería el único alumno que iría solo en el pupitre. Por lo tanto, el número de "pasos" que hará nuestro algoritmo voraz será igual al número de alumnos entre 2. El esquema, que incluye la parte primera de crear la matriz "suma" (sería el pretratamiento de datos), queda (nótese que MAX_ALUMNOS es 100, según se indica en la especificación del problema, y que N es el número de alumnos):

```
Algortimo_voraz(){
    solución: array[1..MAX_ALUMNOS] de entero;
    bact: entero;
    bact := 0;
    //Inicializamos el array solución.
    para i:=1..MAX_ALUMNOS hacer
          solución[i] := -1;
    finpara
    //Inicializamos la matriz suma.
    para i:=1..N hacer
          para j:=1..N hacer
             suma[j][h]=(amistad[j][h]+amistad[h][j])*(trabajo[j][h]+
                trabajo[h][j]);
             suma[h][j] = suma[j][h];
          finpara
    finpara

    //Ahora, el algoritmo voraz como tal.
    asignados: entero;
    asignados := 0;
    mientras(asignados != N) hacer
```

```
            p: pareja;
            p = seleccionar();
            solución[asignados] = p.x;
            solución[asignados + 1] = p.y;
            asignados = asignados + 2;
            bact := bact + suma[p.x][p.y];
            si (N - asignados == 1) entonces
                  i: entero;
                  i := 0;
                  mientras (i < N && visitado[i]) hacer
                        i := i + 1;
                  finmientras
                  solución[asignados] := i;
                  visitado[i] := true;
                  asignados := asignados + 1;
            finsi
      finmientras
}
```

Nótese que no incluimos la parte de código que muestra por pantalla el orden de los alumnos y el beneficio, pues consideramos que eso no es realmente parte del algoritmo voraz. Además, ya se comentará más adelante, pero hay que notar que aquí se hace uso del valor máximo de alumnos según la especificación del problema, MAX_ALUMNOS, pero luego a la hora de hacer el estudio teórico este valor se tomará como N (número de alumnos) para no tomarlo como constante.

También cabría comentar cómo son las distintas funciones que se incluyen en el esquema general de un algoritmo voraz (solo "seleccionar" se implementa de forma separada como función, las demás serían partes del código del algoritmo voraz). La función "solución" en este caso es simplemente la comprobación que se hace en el bucle principal. Tendremos solución cuando se hayan asignado todos los alumnos, pues en este problema siempre habrá solución (nuestro objetivo realmente es maximizar el beneficio de dicha solución). La función "factible" por su parte

será siempre cierta, siempre podemos sentar a los alumnos. Por lo tanto, siempre se ejecuta "insertar", que simplemente inserta en el array solución a los alumnos correspondientes. Por último, la función seleccionar se trata en el siguiente punto.

En cuanto a las variables globales usadas, nos encontramos con N, que es el número de alumnos; la matriz suma, que se usa sobre todo en la función seleccionar y por ese motivo se decidió ponerla global, para ahorrar problemas de paso por parámetro; y un array de visitados que nos indica qué alumnos se han asignado ya y cuales no, lo cual se hace en la función seleccionar y por tanto se pudo global.

3.1.3 FUNCIÓN SELECCIONAR

Para seleccionar, primero hay que tener en cuenta que no se obtiene, ni se nos pedía, una solución óptima. El criterio que seguimos fue, en cada selección, devolver la pareja con el mayor valor en la matriz "suma", que almacena los beneficios de sentar los alumnos entre sí. Claro está, esto no tiene por qué devolver la solución óptima. El esquema queda:

```
seleccionar(var resultado: pareja){
    x, y, maximo: enteros;
    x := 0;
    y := 0;
    máximo := -1;
    para i:=1..N hacer
        para j:=i+1..N hacer
            si (!visitado[i] Y !visitado[j]) entonces
                si (suma[i][j] > máximo) entonces
                    máximo := suma[i][j];
                    x := i;
                    y := j;
                finsi
            finsi
        finpara
    finpara
```

```
        visitado[x] := true;
        visitado[y] := true;
        resultado: pareja;
        resultado.x := x;
        resultado.y := y;
        devolver resultado;
    }
```

Como se puede observar, la función seleccionar simplemente obtiene para qué alumnos que no hayan sido asignados ya tenemos el mayor beneficio si los sentásemos juntos. También se pone como visitado las posiciones correspondientes en el array visitados.

3.2 ESTUDIO TEÓRICO DEL TIEMPO DE EJECUCIÓN AVANCE RÁPIDO

En este apartado se hará un estudio teórico del tiempo de ejecución del algoritmo voraz a partir del planteamiento en pseudocódigo. Para ello, primero se tratará la función seleccionar para luego ya hacer el estudio para el algoritmo voraz como tal.

3.2.1 SELECCIONAR

Empezamos con el mejor caso. Claramente, observamos que esto será cuando sólo se entre una vez a los condicionales del bucle doble. Esto ocurrirá, por ejemplo, cuando sólo queden por asignar 2 alumnos. Así, en este mejor caso tendremos:

$$t_m(n) = 3 + \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} (1+1) \right) + 4 + 5$$

Nótese que el 4 son las instrucciones que se harán la única vez que se entra en el condicional, por lo tanto, las sumamos directamente fuera del sumatorio. También hay que notar que n es el número de alumnos, pero que en el código la variable que almacenaba este valor era N. Así, se queda:

$$t_m(n) = 12 + \sum_{i=1}^{n} (1 + (n-i) * 2) \; = \; 12 \; + \; n * (2n+1) - \sum_{i=1}^{n} 2i$$
$$= 12 + 2n^2 + n - n(n+1) \; = \; 12 + 2n^2 + n - n^2 - n \; = \; n^2 + 12$$

Por lo tanto, tenemos: $t(n) \in \Omega(n^2)$.

Vamos ahora con el peor caso. Este será cuando entremos siempre en todos los condicionales. Para ello, tendrán que estar todos los alumnos sin asignar (el array visitado todo a false) y tener los valores de la matriz en orden creciente, para siempre ir encontrando un nuevo valor máximo. Así, tendremos:

$$t_M(n) = 3 + \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} (1+5) \right) + 5$$

Desarrollando obtenemos:

$$t_M(n) = 8 + \sum_{i=1}^{n} (1 + (n-i) * 6) = 8 + n * (6n+1) - \sum_{i=1}^{n} 6i = 8 + 6n^2 + n - 3n(n+1)$$

Finalmente tenemos:

$$t_M(n) = 8 + 3n^2 - 2n$$

Por lo que obtenemos que: $t(n) \in O(n^2)$.

Como coinciden ambos ordenes, podemos afirmar que: $t(n) \in \Theta(n^2)$.

2.2.2 ALGORITMO VORAZ

Por comodidad, separaremos este estudio en pretratamiento de datos y el algoritmo voraz en sí. Aunque en el pseudocódigo se usa MAX_ALUMNOS, que según la especificación del problema es 100, para el bucle de inicialización de la solución, para este estudio teórico vamos a tomar ese bucle de tamaño n, es decir, de tamaño el número de alumnos, y así podremos tener una visión mucho más general del tiempo de ejecución de este algoritmo. El pretratamiento supone entonces:

$$g(n) = 1 + 2 * n + \sum_{i=1}^{n} (1 + \sum_{j=1}^{n} (1+2)) = 1 + 2 * n + n * (1 + 3n) = 1 + 3n + 3n^2$$

Por lo tanto, teniendo en cuenta lo comentado anteriormente, tendremos que el pretratamiento de datos supone: $g(n) \in \Theta(n^2)$

Ahora, veamos el algoritmo voraz. Nótese cómo, según el número de alumnos sea par o impar, se tendrá que quedar un alumno sólo o no. Esto implica que según el número de alumnos sea par o impar, se ejecutará el condicional del bucle o no, pero veremos que esto no supone ningún cambio en el orden de ejecución. Así, si el número de alumnos es par, el algoritmo voraz (sin contar el pretratamiento de datos):

$$f(n) = 1 + \sum_{i=1}^{n/2} (c * n^2 + 4 + 1 + 1) + 1$$

Nótese que como seleccionar obtuvimos que tenía orden exacto, podemos usar un múltiplo de ese orden. Además, también hay que tener en cuenta que, como estamos suponiendo que el número de alumnos es par, no se entra en el condicional que hay dentro del bucle. También hay que observar que en cada iteración la variable que determina el número de iteraciones se aumenta en dos unidades, por lo que el sumatorio va desde 1 hasta n/2. Si desarrollamos la expresión anterior, obtenemos:

$$f(n) = 1 + \frac{n}{2}(c * n^2 + 4 + 1 + 1) + 1 \ = 2 + \frac{nc}{2} + \frac{n^3}{2} + 2n + n$$

Por lo tanto, obtenemos que, si el número de alumnos es par, el tiempo de ejecución cumple: $f(n) \ \in \Theta(n^3)$

Veamos si el número de alumnos es impar. En este caso, el número de iteraciones del bucle es el mismo, solo que en la última se entrará al condicional y se ejecutarán sus instrucciones. No obstante, en este caso, a la hora de buscar el alumno que no ha sido emparejado aún, si que tendremos que distinguir entre un mejor y un peor caso, pues lo mejor será encontrar ese alumno a la primera y lo peor que sea el último. Sin embargo, ya veremos que esto no afecta al orden final. En el mejor caso, si el alumno que falta por emparejar es el primero:

$$f(n) = 1 + \sum_{i=1}^{n/2} (c * n^2 + 4 + 1 + 1) + 1 + \ 1 \ + 1 + 3 = 1 + \frac{n}{2}(c * n^2 + 4 + 1) + 5$$

$$= \ 7 + \frac{nc}{2} + \frac{n^3}{2} + 2n + n$$

Si consideramos el peor caso, que el alumno que falta por emparejar sea el último:

$$f(n) = 1 + \sum_{i=1}^{n/2}(c * n^2 + 4 + 1 + 1) + 1 + 1 + \sum_{i=1}^{n-1} 2 + 1 + 3$$

$$= 1 + \frac{n}{2}(c * n^2 + 4 + 2) + 2 * (n - 1) + 6 = 5 + \frac{nc}{2} + \frac{n^3}{2} + 2n + 3n$$

Observamos que ambos casos obtenemos que: $f(n) \in \Theta(n^3)$

Así, vemos que sea el número de alumnos par o impar, llegamos a la misma conclusión:

$f(n) \in \Theta(n^3)$

Por lo tanto, vemos que aquí el orden de complejidad mayor lo tendrá el algoritmo voraz y no el pretratamiento de datos. Así, todo supondrá un coste de orden: $\Theta(n^3)$

3.3 PROGRAMACIÓN DEL ALGORITMO VORAZ

```
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;

#define MAX_ALUMNOS 100
////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////

int N;
int amistad[MAX_ALUMNOS][MAX_ALUMNOS];
int trabajo[MAX_ALUMNOS][MAX_ALUMNOS];
int suma[MAX_ALUMNOS][MAX_ALUMNOS];
bool visitado[MAX_ALUMNOS];

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////

//Usamos un struct pareja para que seleccionar pueda coger de 2 en 2
struct pareja{
```

```
    int x;
    int y;
};

//Seleccionar busca la pareja de no asignados con mayor valor en la matriz suma
pareja seleccionar(){
    int x = 0;
    int y = 0;
    int maximo = -1;
    for (int i = 0; i < N; i++){
        for (int j = i+1; j < N; j++){
            if (!visitado[i] && !visitado[j]){
                if (suma[i][j] > maximo){
                    x = i;
                    y = j;
                    maximo = suma[i][j];
                }
            }
        }
    }
    visitado[x] = true;
    visitado[y] = true;

    pareja resultado;
    resultado.x = x;
    resultado.y = y;
    return resultado;
}

void algoritmo_voraz(){
    int solucion[MAX_ALUMNOS];
    int bact = 0;

    for (int j = 0; j < MAX_ALUMNOS; j++){
        solucion[j] = -1;
```

```
    }

    //Creamos la matriz "suma", con la que realmente trabajamos.
    //Almacena los beneficios calculados según se indican en la especificación.
    for (int j = 0; j < N; j++){
        for (int h = j; h < N; h++){
            suma[j][h] = (amistad[j][h]+amistad[h][j])*(trabajo[j][h]+trabajo[h][j]);
            suma[h][j] = suma[j][h];
        }
    }

    int asignados = 0;
    //Mientras no asignemos todos, no hay SOLUCIÓN.
    while(asignados != N){
        pareja p = seleccionar();
        //SIEMPRE INSERTAMOS, FACTIBLE es siempre cierto.
        solucion[asignados] = p.x;
        solucion[asignados + 1] = p.y;
        asignados = asignados + 2;
        //Actualizamos el beneficio
        bact = bact + suma[p.x][p.y];
        //Si son impares y queda uno suelto, lo dejamos solo (será el único).
        if (N - asignados == 1){
            int i = 0;
            while (i < N && visitado[i]){
                i++;
            }
            solucion[asignados] = i;
            visitado[i] = true;
            asignados++;
        }
    }

    int i;
    cout << bact << endl;
```

```
    cout << solucion[0];
    i = 1;
    while (i < N){
        cout << " " << solucion[i];
        i++;
    }
    cout << endl;

}
```

No se muestra el código del main, pues no queremos que se extienda excesivamente esta sección de código y en este caso es un método principal muy sencillo sin nada especial. No obstante, se puede encontrar en los archivos de código entregados junto a esta memoria.

En cuanto a las variables globales, vemos cómo se pusieron las matrices como globales para agilizar el trabajo con las mismas en diferentes funciones del programa. También se usa un array de visitados para controlar que alumnos han sido asignados a un pupitre y cuáles no, además de la variable N, de tipo entero, que almacena el número de alumnos.

3.4 ESTUDIO EXPERIMENTAL DEL TIEMPO DE EJECUCIÓN

Para el estudio experimental, aunque no se pedía en la especificación de la práctica, implementamos un generador de casos de prueba promedios, es decir, aleatorios. Lo hicimos así pues, como se ha podido ver en el estudio teórico del tiempo de ejecución, obtenemos un orden exacto, por lo que la diferencia entre el mejor caso y el peor puede ser casi inapreciable. Por lo tanto, simplemente medimos el tiempo en casos aleatorios, promedio. El generador básicamente genera dos matrices de tamaño nxn (donde n es el número de alumnos) teniendo en cuenta que la diagonal debe ser 0 y que los números deben estar entre 1 y 9, ambos inclusive. A continuación, se muestra el código del generador, aunque el fichero .cpp se entrega con la práctica:

```
#include "generador.hpp"
#include <stdio.h>
#include "math.h"
#include <iostream>
#include <string>
```

```
#include <stdlib.h>

using namespace std;



//En el caso promedio se generan cadenas aleatorias.
void genera_caso_promedio (int n, int ** amistad, int ** trabajo){
    srand(time(NULL) + rand());
    int random;
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            if (i == j){
                amistad[i][j] = 0;
                trabajo[i][j] = 0;
            } else{
                random = rand()%9 + 1;
                amistad[i][j] = random;
                random = rand()%9 + 1;
                trabajo[i][j] = random;
            }


        }

    }
    return;
}
```

Vemos como este generador hace uso de punteros para crear las matrices necesarias. Esto es así pues para realizar el estudio experimental, decidimos obviar las restricciones en cuanto a tamaño que se plantean en la especificación del problema, para así poder hacer un estudio un poco más completo. Por ello, no podíamos tener un tamaño fijo de matrices y tuvimos que hacer uso de punteros. Además, esto también permite declarar las matrices en el main y modificarlas en los generadores sin problema alguno.

Para la medición del tiempo era imprescindible ser capaces de hacerlo sin contar todo lo que supone crear las matrices, inicializarlas, etc. Por lo tanto, tras investigar un poco por internet, decidimos hacer uso de la biblioteca "chrono" para medir el tiempo transcurrido entre que se entra en el algoritmo voraz y se termina y se procede a imprimir por pantalla el resultado pedido. Este es el tiempo que nos interesa y el que se mostraba por pantalla durante el estudio experimental de nuestro programa. Nótese que sí que se incluye en el tiempo medido el tiempo que se tarda en inicializar el array donde se almacena la solución y en crear la matriz suma (que es con la que realmente trabajamos). Esto lo hicimos así pues consideramos que esto si formaba parte del "pretratamiento de datos" necesario para la implementación de este algoritmo voraz, al contrario que la parte de código que genera las otras matrices y las inicializa. El algoritmo voraz queda de la siguiente manera cuando le incluimos la medida de tiempos:

```
void algoritmo_voraz(){
    auto t0 = std::chrono::high_resolution_clock::now();

    int * solucion;
    solucion = new int[N];
    int bact = 0;
    for (int j = 0; j < N; j++){
        solucion[j] = -1;
    }

    for (int j = 0; j < N; j++){
        for (int h = j; h < N; h++){
            suma[j][h] =
(amistad[j][h]+amistad[h][j])*(trabajo[j][h]+trabajo[h][j]);
            suma[h][j] = suma[j][h];
        }
    }

    int asignados = 0;
    while(asignados != N){
        pareja p = seleccionar();
        solucion[asignados] = p.x;
        solucion[asignados + 1] = p.y;
        asignados = asignados + 2;
```

```
            bact = bact + suma[p.x][p.y];
            if (N - asignados == 1){
                int i = 0;
                while (i < N && visitado[i]){
                    i++;
                }
                solucion[asignados] = i;
                visitado[i] = true;
                asignados++;
            }
        }

        auto t1 = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0);
        cout << "Execution time: " << time.count()*1e-9 << endl;

        cout << bact << endl;
        cout << solucion[0];
        i = 1;
        while (i < N){
            cout << " " << solucion[i];
            i++;
        }
        cout << endl;

}
```

Para llevar a cabo este estudio experimental se hicieron 20 ejecuciones para diferentes tamaños. Se tomaron medidas para n = 10, n = 25, n = 50, n =75, n = 100, n = 125, n = 150, n = 175 y n = 200. Decidimos no tomar medidas para valores mayores porque la gráfica que obtendríamos, debido a la diferencia de valor entre los tiempos para n grandes y los tiempos para n pequeñas, haría que estos tiempos pequeños no se distinguiesen bien en la escala.

Así, a continuación, se puede encontrar la gráfica obtenida. En el eje Y se muestra el tiempo y en el eje X el número de alumnos. La línea de tendencia se muestra en color rojo. Cabe mencionar

que con la presente memoria se entrega también el archivo Excel donde figuran los datos con los que se creó la gráfica y la propia gráfica.

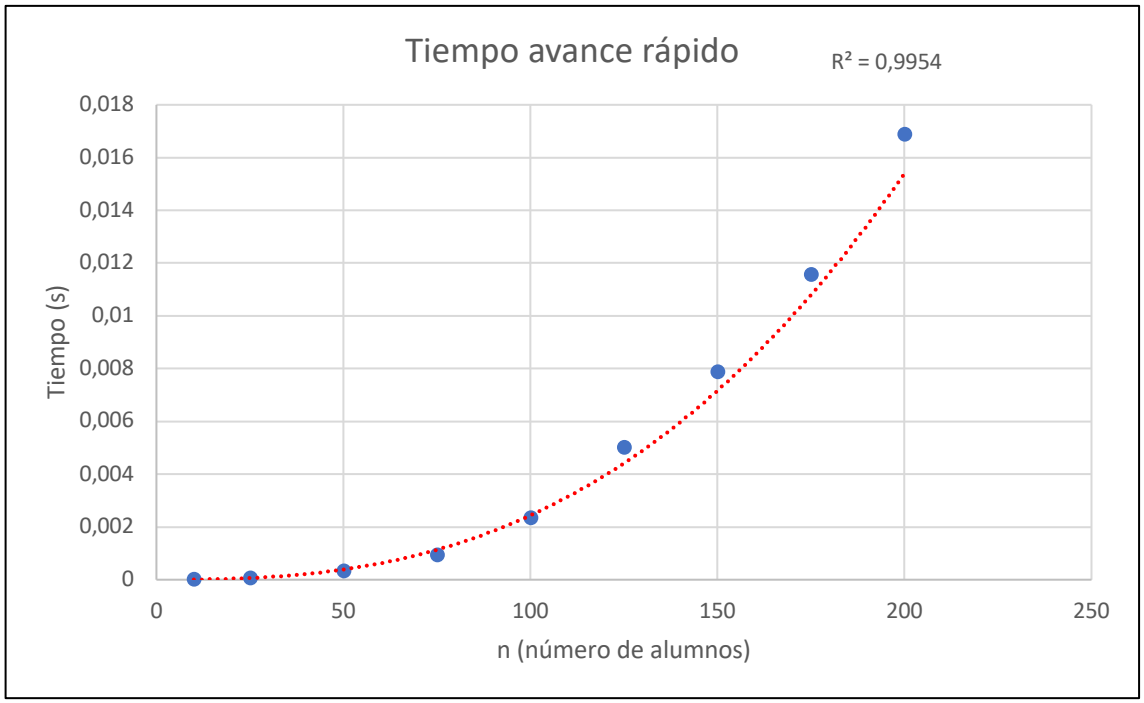

Figura 1. Gráfica para los tiempos de ejecución del algoritmo voraz para diferentes valores de n. Fuente: propia.

Se observa cómo la línea de tendencia mostrada en color rojo se ajusta muy bien al crecimiento de los datos, ya que el valor de R^2 es muy cercano a 1. La línea de tendencia es potencial, lo cual nos indica que estos datos tienen un crecimiento potencial. La comparación de este resultado con los obtenidos de forma teórica se hará en el siguiente apartado.

3.5 CONTRASTE ENTRE EL ESTUDIO TEÓRICO Y EL EXPERIMENTAL

En el estudio experimental, como se puede observar en la Figura 1 mostrada anteriormente, se obtuvo que el tiempo crece de forma potencial según se aumenta n. Esto significa que "sigue" una función del tipo x elevado a un número real. Haciendo uso de Excel, mostramos a continuación la misma gráfica anterior, pero mostrando también la ecuación que sigue la línea de tendencia:

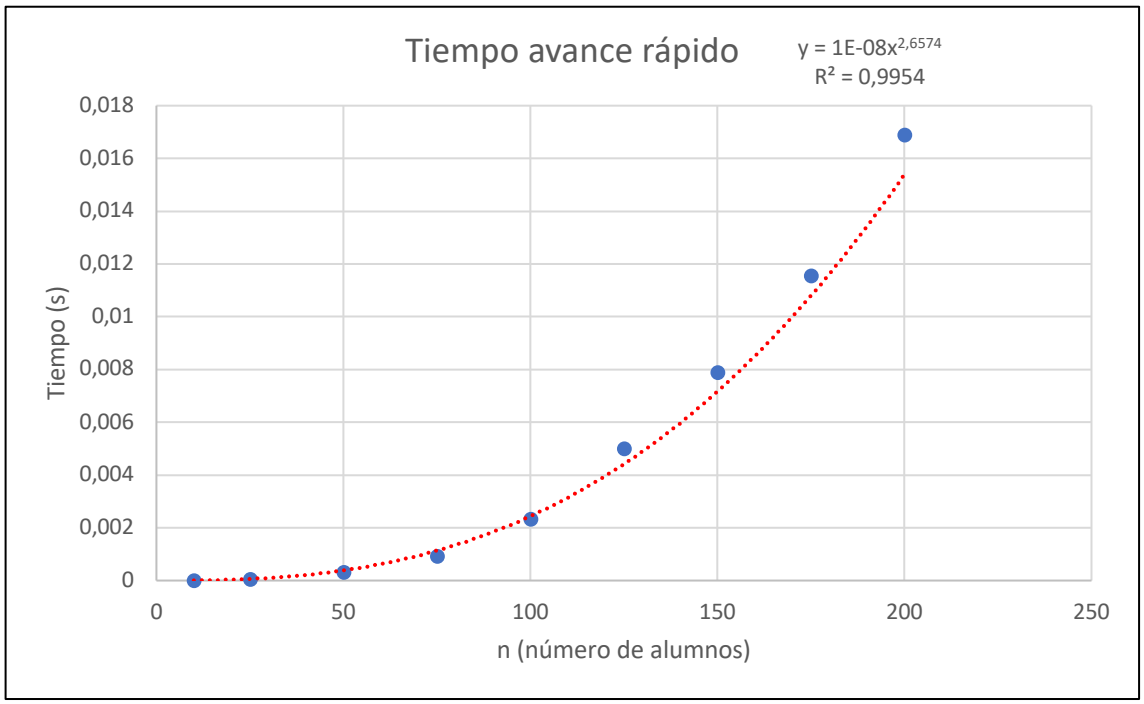

Figura 2. Gráfica para los tiempos de ejecución del algoritmo voraz para diferentes valores de n y la ecuación de la línea de tendencia que mejor se acopla. Fuente: propia.

Como se observa, la ecuación que sigue la línea de tendencia mostrada es un múltiplo de $x^{2.63}$. En el estudio teórico del tiempo de ejecución, obtuvimos que el tiempo de ejecución pertenecía al orden exacto de n^3, por lo que lo ideal hubiese sido que la línea de tendencia siguiese una ecuación múltiplo de x^3. No obstante, el valor obtenido está muy cerca y confirma que, efectivamente, el algoritmo voraz planteado crece potencialmente con el aumento del número de alumnos. La diferencia obtenida en el exponente de la ecuación puede ser debida a diversos factores, como por ejemplo el hecho de que el estudio teórico realizado ha sido haciendo uso del "conteo de instrucciones", lo cual no refleja que algunas instrucciones tienen un coste mayor que otras, entre otros posibles factores.

Por lo tanto, consideramos que pese a la pequeña diferencia obtenida, el estudio experimental confirma el resultado obtenido de forma teórica.

4. BACKTRACKING

Como se indicó en la introducción, nos tocó solucionar el problema F de backtracking.

4.1 PSEUDOCÓDIGO

Primero, cabe mencionar que en muchos casos se trabaja con variables que no se declaran, pues son variables globales. Entre éstas encontramos: nivel (nivel del árbol en el que nos encontramos), M (dinero disponible), C (nº de prendas), pact (dinero gastado con la solución parcial), precios (matriz con los precios de cada modelo de cada prenda), s (array con la solución parcial) y modelos (array que almacena el número de modelos disponibles para cada prenda).

4.1.1 FUNCIÓN GENERAR

Como vimos en clase de teoría, generar siempre genera. En esta función se aumenta el valor de la solución para un nivel dado (se coge el siguiente modelo de una prenda). También se actualiza el valor de pact, la variable que lleva la cuenta del dinero gastado con la solución parcial. Además, hay que comprobar si nos encontramos en el primer nodo de un nivel pues entonces no debemos restar nada, pero si no es así habrá que restar el precio del modelo anterior y sumar el del nuevo modelo

```
generar(){
    si (s[nivel] != 0) entonces
            pact := pact – precios[nivel][s[nivel]];
            s[nivel] := s[nivel] + 1;
            pact := pact + precios[nivel][s[nivel]];
    sino
            s[nivel] := s[nivel] + 1;
            pact := pact + precios[nivel][s[nivel]];
    finsi
}
```

4.1.2 FUNCIÓN SOLUCIÓN

```
solucion(){
    devolver (nivel == C) Y (pact <= M);
}
```

Como vemos, estaremos ante una posible solución si estamos en el último nivel (hemos cogido un modelo de cada prenda) y el dinero gastado con esas elecciones no supera la disponible.

4.1.3 FUNCIÓN CRITERIO

```
criterio(int voa){
    si (nivel == C OR pact >= M OR pact + precios[nivel+1][0] <= voa) entonces
        devolver false;
    finsi
    si (pact + precios[0][nivel + 1] > M) entonces
        devolver false;
    finsi
    si (voa == M) entonces
        devolver false;
    finsi
    devolver true;
}
```

En este método hay que destacar las podas que se realizan. Obviamente, si ya hemos llegado al último nivel, el dinero gastado es mayor o igual que el disponible o la mejor solución ya encontrada es la mejor posible, no es necesario seguir recorriendo el árbol. Pero en esta función se incluyen dos condiciones más. Como para evitar problemas al usar el nivel actual en la matriz precios, se decidió crearla con una columna y una fila de más para poder usar desde la fila y columna 1, se usó la columna 0 para ir almacenando el dinero máximo que se podría gastar en comprar las prendas del nivel correspondiente en adelante. Es decir, en precios[j][0] se encuentra el dinero máximo que nos podemos gastar en comprar un modelo de la prenda j, otro de la j+1, y así sucesivamente hasta la última. Por otro lado, en la fila 0 encontramos lo contrario, el mínimo dinero que tendremos que gastarnos en comprar la prenda del nivel correspondiente y las siguientes. Es decir, en precios[0][j] encontramos el mínimo dinero que habrá que gastarse en comprar un modelo de la prenda j, otro de la j+1 y así sucesivamente hasta la última prenda. Con esto, se introdujeron dos podas: si, con las prendas compradas con la solución parcial, lo máximo que nos podemos gastar no supera la mejor solución ya encontrada, no es necesario que sigamos recorriendo esa rama. Además, si con las prendas ya compradas, lo mínimo que nos gastaremos

en comprar un modelo de cada prenda supera el dinero que nos podemos gastar en total, tampoco es necesario seguir recorriendo dicha rama pues sabemos que no habrá solución.

4.1.4 FUNCIÓN MAS_HERMANOS

```
mas_hermanos(){
    devolver s[nivel] < modelos[nivel];
}
```

En este método se indica si hay más hermanos en el árbol. Esto será cierto si el modelo que hemos cogido para el nivel actual no es el último, es decir, es menor que el número de modelos disponible para esa prenda.

4.1.5 FUNCIÓN RETROCEDER

```
retroceder(){
    pact := pact – precios[nivel][s[nivel]];
    s[nivel] := 0;
    nivel = nivel – 1;
}
```

En este se realiza el retroceso, que se hará cuando no se puedan generar más hermanos. Para ello, se resta el precio del modelo de la prenda del nivel del que queremos retroceder, se pone a 0 la solución para dicho nivel, y se retrocede.

4.1.5 BACKTRACKING

```
Backtracking(){
    nivel := 1;
    pact := 0;
    voa: entero;
    voa := -1;
    hacer
        generar();
        si (solucion() Y pact > voa) entonces
            voa := pact;
```

```
            finsi
            si (criterio(voa)) entonces
                  nivel = nivel + 1;
            sino
                  mientras ((!mas_hermanos() || voa == M) Y nivel > 0) hacer
                        retroceder();
                  finmientras
            finsi
      hasta (nivel == 0);
      si (voa == -1) entonces
            imprimir("No solution");
      sino
            imprimir(voa);
      finsi
}
```

Este método es el principal del esquema de backtracking. Sigue el esquema explicado en clase de teoría, aunque nosotros le hemos añadido algún detalle. Por ejemplo, para que el mejor caso sea más eficiente, a la hora de retroceder si no hay más hermanos, se añade la opción de retroceder si la mejor solución encontrada es ya la máxima posible, por lo que no es necesario visitar los hermanos en caso de que hubiera.

4.2 ESTUDIO TEÓRICO DEL TIEMPO DE EJECUCIÓN BACKTRACKING

Observando el pseudocódigo del algoritmo, es claro que todas las funciones, excepto la función principal del backtracking, supondrán un esfuerzo de orden constante. Por lo tanto, podemos estudiar el tiempo de ejecución de nuestro algoritmo de backtracking tomando el coste de estas funciones como constante. Comenzamos con el mejor caso.

Primero, cabe comentar que vamos a considerar el mejor caso cuando hay solución, pues podríamos decir que, si el dinero disponible para comprar es menor que el número de prendas, directamente podemos asegurar que no hay solución. Pero consideraremos el caso en que si hay solución. En este caso, lo mejor que podría pasar es que se encontrase una solución que gastase

el máximo dinero disponible a la primera. Por lo tanto, se irán generando nodos hasta llegar al último nivel, donde confirmaremos que es una solución (que será la máxima posible). Entonces, al volver, tendría que ir generando los hermanos de cada nivel. Por esta razón, a la comprobación que se hace para ver si se retrocede, le añadimos otra opción: aunque haya hermanos, si la mejor solución encontrada ya es la mejor posible, no es necesario generar el hermano. Con esto, este mejor caso es más eficiente, pues el algoritmo simplemente irá generando los niveles hasta el último, y luego tendrá que ir retrocediendo hasta llegar al primer nivel. Como ya se ha encontrado la mejor solución, criterio será falso y se tendrá que retroceder al nivel 0, terminando la ejecución. Tendremos entonces (tomando n como el número de prendas (en el algoritmo es la C)):

$$t_m(n) = 3 + \sum_{i=1}^{n-1}(1 + c + 1 + s + 1 + d + 1) + c + 1 + s + 1 + d + \sum_{i=n}^{1}(1 + h + r) + 1 + 1$$

En las cuentas anteriores, la constante c es el tiempo de *generar*, la s el de *solución,* la d el de criterio, la h de *mas_hermanos* y la r de *retroceder*. Nótese que hay que sumar 1 en cada iteración del primer sumatorio por la comprobación del "*hacer hasta*". También hay que sumar dos instrucciones más al final, una de la última comprobación de "mientras" del retroceso y la otra de la última comprobación del "*hacer hasta*". De la expresión anterior se obtiene:

$$t_m(n) = 3 + (n-1) \times (4 + s + d + c) + 2 + c + s + d + n(1 + h + r) + 2$$

Como vemos en la expresión anterior, es claro que $t(n) \in \Omega(n)$.

En cuanto al peor caso, este será cuando se tenga que generar el árbol completo, que se de el caso de que nunca se produzca ninguna poda, que no se encuentre la mejor solución posible, que nunca tengamos una solución parcial que no cumpla los criterios y podamos dejar de bajar por el árbol, etc. Como en nuestro problema cada nivel puede tener un número de hijos diferente (cada prenda puede tener un número de modelos distinto), el número de nodos será:$\sum_{i=1}^{n}(\prod_{j=1}^{i} m_i)$, donde n es el número de prendas y m_i es el número de modelos disponibles para la prenda i. Como hemos visto que, claramente, las funciones del algoritmo de backtracking suponen un esfuerzo de orden constante, es claro que recorrer todos los nodos supondrá un orden igual al número de nodos, que es el número total de nodos mostrado. Entonces tenemos que:

$$t_M(n, m_1, m_2, \ldots, m_n) \in O(\sum_{i=1}^{n}(\prod_{j=1}^{i} m_i))$$

4.3 PROGRAMACIÓN DEL ALGORITMO DE BACKTRACKING

```
#include <stdlib.h>
#include <string.h>
#include <iostream>
using namespace std;

#define MAX_PRENDAS 20
#define MAX_MODELOS 20
////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////

int nivel;
int M;
int C;
int precios[MAX_PRENDAS + 1][MAX_MODELOS + 1];
int s[MAX_PRENDAS + 1];
int modelos[MAX_PRENDAS + 1];
int pact;

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////

//Función generar del esquema de backtracking
void generar(){
    //Si no es 0 restamos antes de sumar
    if (s[nivel] != 0){
        pact = pact - precios[nivel][s[nivel]];
        s[nivel] = s[nivel] + 1;
        pact = pact + precios[nivel][s[nivel]];
    } else {
        s[nivel] = s[nivel] + 1;
        pact = pact + precios[nivel][s[nivel]];
    }
```

```
}


//Función solucion del esquema de backtracking
bool solucion(){
    //Si estamos en el ultimo nivel y el dinero gastado no es demasiado
devolvemos true.
    return (nivel == C) && (pact <= M);
}


//Función criterio del esquema de backtracking
//Separamos en varios condicionales para que no sea un condicional muy largo.
bool criterio(int voa){
    if ((nivel == C) || (pact >= M) || (pact + precios[nivel + 1][0] <= voa)){
        return false;
    }
    if (pact + precios[0][nivel + 1] > M){
        return false;
    }
    if (voa == M){
        return false;
    }
    return true;
}


//Función mas_hermanos del esquema de backtracking
bool mas_hermanos(){
    //devolvemos true si no es el último modelo.
    return s[nivel] < modelos[nivel];


}


//Función retroceder del esquema de backtracking
void retroceder(){
    //Hay que restar un poner a 0 el array s antes de retroceder en el nivel.
    pact = pact - precios[nivel][s[nivel]];
```

```
    s[nivel] = 0;
    nivel--;
}

//Función principal del esquema de backtracking
void backtracking(){
    nivel = 1;
    pact = 0;
    int voa = -1;
    do {
        generar();
        if ((solucion()) && (pact > voa)){
            voa = pact;
        }
        if (criterio(voa)){
            nivel++;
        } else {
            //Añadimos la condición de que si la sol. encontrada es la maxima
no se visitan los hermanos, no es necesario.
            while ((!mas_hermanos() || voa == M) && (nivel > 0)){
                retroceder();
            }
        }
    } while (nivel > 0);
    if (voa == -1){
        cout << "no solution" << endl;
    } else {
        cout << voa << endl;
    }
}
```

Vemos que tampoco se ha incluido en este caso el código del main, pues no queríamos hacer demasiado extensa esta sección, pero se puede encontrar en el archivo .cpp entregado junto a esta memoria. En cuanto a las variables globales usadas, vemos que son las mismas que se indicaron al comienzo de la sección de pseudocódigo. Se tomaron como globales para facilitar su

uso y actualización en varias funciones del código sin tener que estar pasándolas como parámetro usando punteros.

4.4 ESTUDIO EXPERIMENTAL DEL TIEMPO DE EJECUCIÓN

Para realizar el estudio experimental, aunque no se pide en la práctica, decidimos implementar un generador. Para el mejor caso, el generador rellena las matrices de forma que, si tomamos el primer modelo de cada prenda, la suma sea justo el dinero máximo que se puede gastar. El resto de la matriz es aleatoria. Cabe mencionar que, como se comentó en el estudio teórico, vamos a tomar el mejor caso como aquel en el que la mejor solución posible se encuentra a la primera, no se tendrá en cuenta la posibilidad de que no haya solución porque todos los modelos de la primera prenda sean demasiado caros y no se profundice más allá del primer nivel, por ejemplo (esto podría ser más rápido que el mejor caso considerado). Por este motivo, para que todo sea correcto, el dinero que se pase como parámetro debe ser mayor que el número de prendas. Además, para que el estudio experimental tenga sentido, en el caso promedio (se generará una matriz aleatoria) hay que tener ciertas consideraciones importantes. Si simplemente generásemos una matriz aleatoria con precios entre 1 y 200 (se decidió mantener esta restricción de la especificación), es bastante probable que no haya solución en la mayoría de los casos si el dinero lo mantenemos con la restricción de que debe estar entre 1 y 200 (si de media los números generados valen 100 se gastará el dinero muy rápido), lo que podría llevar a que en bastantes casos el caso promedio saliese más rápido que el mejor caso. Para evitar esto, obviamos la restricción del dinero, y tomamos el dinero como la mitad del máximo que podríamos gastarnos. Es decir, si son 20 prendas y sabemos que como máximo cada una valdrá 200, tomaremos como dinero 2000. Así, hay más posibilidades de que haya solución y será un estudio más realista. Tomamos esta manera de elegir el dinero pues sería como suponer que cada prenda, de media, cuesta 100 euros, lo cual creemos que tiene sentido si recordamos que los números se generan aleatoriamente. Además, esto tiene la ventaja de que todas las podas implementadas tienen posibilidad de producirse, lo cual es adecuado si queremos que sea un estudio experimental del tiempo promedio. Para el mejor caso, se usó siempre 200 como dinero disponible, pero esto realmente da igual, siempre y cuando sea un número mayor al número de prendas. En cuanto al precio de cada modelo, decidimos mantener la acotación que se hace en la especificación porque al trabajar con números generados de forma aleatoria, de alguna manera hay que poner un límite a esos valores para que no salgan valores exageradamente grandes, por lo que decidimos mantener el mismo criterio para simplificar la implementación. A continuación, se muestra el generador:

```
#include "generadorBT.hpp"

#include <stdio.h>

#include "math.h"

#include <iostream>

#include <stdlib.h>


using namespace std;


//En el mejor caso coger el primer modelo de cada prenda supone gastar justo el dinero que tenemos

void genera_caso_mejor (int nprendas, int * nmodelos, int ** precios, int dinero_total){

    int dinero_gastado = 0;

    for (int i = 1; i < nprendas; i++){

        precios[i][1] = dinero_total/nprendas;

        precios[i][0] = precios[i][1];

        precios[0][i] = precios[i][1];

        dinero_gastado = dinero_gastado + precios[i][1];

    }

    precios[nprendas][1] = dinero_total - dinero_gastado;

    precios[nprendas][0] = precios[nprendas][1];

    precios[0][nprendas] = precios[nprendas][1];

    srand(time(NULL) + rand());

    int random;

    for (int i = 1; i <= nprendas; i++){

        int j = 2;

        int maximo = precios[i][0];

        int minimo = precios[0][i];

        while (j <= nmodelos[i]){

            random = rand()%200 + 1;

            precios[i][j] = random;

            if (random > maximo){

                maximo = random;

            }

            if (random < minimo){
```

```
                minimo = random;
            }
            precios[i][0] = maximo;
            precios[0][i] = minimo;
            j++;
        }
    }
    for (int i = nprendas-1; i > 0; i--){
        precios[i][0] = precios[i][0] + precios[i+1][0];
        precios[0][i] = precios[0][i] + precios[0][i+1];
    }
    return;
}



//En el caso promedio se generan matrices aleatorias.
void genera_caso_promedio (int nprendas, int * nmodelos, int ** precios){
    srand(time(NULL) + rand());
    int random;
    for (int i = 1; i <= nprendas; i++){
        int j = 1;
        int maximo = 0;
        int minimo = 999999;
        while (j <= nmodelos[i]){
            random = rand()%200 + 1;
            precios[i][j] = random;
            if (random > maximo){
                maximo = random;
            }
            if (random < minimo){
                minimo = random;
            }
            precios[i][0] = maximo;
            precios[0][i] = minimo;
```

```
            j++;
        }
    }
    for (int i = nprendas-1; i > 0; i--){
        precios[i][0] = precios[i][0] + precios[i+1][0];
        precios[0][i] = precios[0][i] + precios[0][i+1];
    }
    return;
}
```

Vemos como estos generadores hacen uso de punteros para crear las matrices necesarias. Esto es así pues para realizar el estudio experimental, decidimos obviar las restricciones en cuanto al número de prendas, número de modelos y dinero disponible que se plantean en la especificación del problema, para así poder hacer un estudio un poco más completo. Por ello, no podíamos tener un tamaño fijo de matrices y tuvimos que hacer uso de punteros. Además, esto también permite declarar las matrices en el main y modificarlas en los generadores sin problema alguno.

Para medir el tiempo de ejecución, de nuevo, se optó por usar la librería <chrono>, y tomar el tiempo antes de empezar el backtracking y después, justo antes de mostrar por pantalla el resultado obtenido (se consideró que esto no forma parte del algoritmo como tal y no debería tenerse en cuenta en la medida del tiempo). El algoritmo de backtracking quedó así:

```
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include "generadorBT.hpp"
#include <chrono>

using namespace std;

#define MAX_PRENDAS 20
#define MAX_MODELOS 20
////////////////////////////////////////////////////////////
////////////        VARIABLES GLOBALES        ///////////////
////////////////////////////////////////////////////////////

int nivel;
```

```
int M;
int C;
int ** precios;
int * s;
int * modelos;
int pact;

////////////////////////////////////////////////////////////
////////////        PROGRAMA PRINCIPAL        ///////////////
////////////////////////////////////////////////////////////


void generar(){
    if (s[nivel] != 0){
        pact = pact - precios[nivel][s[nivel]];
        s[nivel] = s[nivel] + 1;
        pact = pact + precios[nivel][s[nivel]];
    } else {
        s[nivel] = s[nivel] + 1;
        pact = pact + precios[nivel][s[nivel]];
    }

}

bool solucion(){
    return (nivel == C) && (pact <= M);
}

bool criterio(int voa){
    if ((nivel == C) || (pact >= M) || (pact + precios[nivel + 1][0] <= voa)){
        return false;
    }
    if (pact + precios[0][nivel + 1] > M){
        return false;
    }
```

```
    if (voa == M){
        return false;
    }
    return true;
}

bool mas_hermanos(){
    return s[nivel] < modelos[nivel];

}

void retroceder(){
    pact = pact - precios[nivel][s[nivel]];
    s[nivel] = 0;
    nivel--;
}

void backtracking(){

    auto t0 = std::chrono::high_resolution_clock::now();
    nivel = 1;
    pact = 0;
    int voa = -1;
    do {
        generar();
        if ((solucion()) && (pact > voa)){
            voa = pact;
           for (int i = 1; i <= C; i++){
              cout << s[i] << " ";
           }
           cout << endl;
        }
        if (criterio(voa)){
            nivel++;
        } else {
```

```
            while ((!mas_hermanos() || voa == M) && (nivel > 0)){
                retroceder();
            }
        }
    } while (nivel > 0);

    auto t1 = std::chrono::high_resolution_clock::now();
    auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0);
    cout << "Execution time: " << time.count()*1e-9 << endl;

    if (voa == -1){
        cout << "no solution" << endl;
    } else {
        cout << voa << endl;
    }
}
```

Para hacer el estudio experimental del mejor caso se hicieron 20 ejecuciones para diferentes tamaños. Se tomaron medidas para n = 5, n = 10, n = 20, n = 40, n = 50, n =60, n = 70 y n = 80. Decidimos no tomar medidas para valores mayores porque, como se podrá ver más adelante, para el peor caso no se toman medidas para valores más grandes debido a los tiempos altos que se obtienen, y queríamos mantener una coherencia entre ambos estudios. Además, se fijó el número de modelos de cada prenda, pues en este caso no debería influir. Decidimos que, en las pruebas a realizar, cada prenda tendría 5 modelos disponibles.

Los resultados obtenidos para el mejor caso se muestran en la siguiente figura. El archivo Excel en el que se encuentran los datos usados y la gráfica se incluye con la entrega.

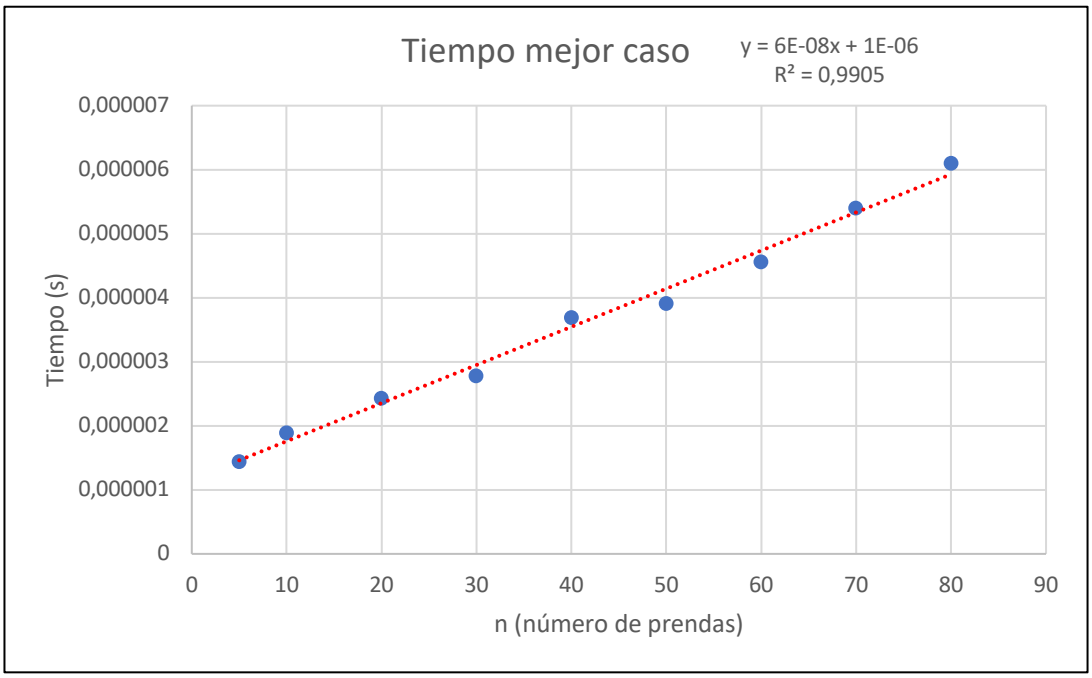

Figura 3. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el mejor caso para diferentes valores del número de prendas. Fuente: propia.

Como se puede observar, se ha incluido una línea de tendencia lineal en la gráfica mostrada. Vemos cómo el valor de R^2 es muy alto, muy cercano a 1, lo que confirma que según nuestras pruebas experimentales el algoritmo en el mejor caso crece de forma lineal según el número de prendas.

Para el caso promedio, como no pudimos, debido a la dificultad que supondría, obtener de forma teórica su orden de ejecución, sólo sabemos que deberá estar acotado por arriba por el peor caso y por el mejor por debajo. Como en el peor caso también influye el número de modelos de cada prenda, para comprobar el posible efecto que tienen estos valores en el caso promedio, decidimos hacer mediciones para diferentes valores de n (número de prendas), y para cada una, poner diferentes números de modelos, pero en todas las prendas el mismo. Es decir, se probó con todas las prendas con 10 modelos o todas con 15 por ejemplo. Esto lo hicimos para que fuese más fácil introducir por pantalla estos valores, copiando y pegando. Así, se probaron los siguientes valores: n = 5, n = 10, n = 20, n = 30, n = 40, n = 50, n =60 , n = 70 y n = 80; y numero de modelos = 5, 10 y 15.

Así, se obtuvieron las siguientes gráficas:

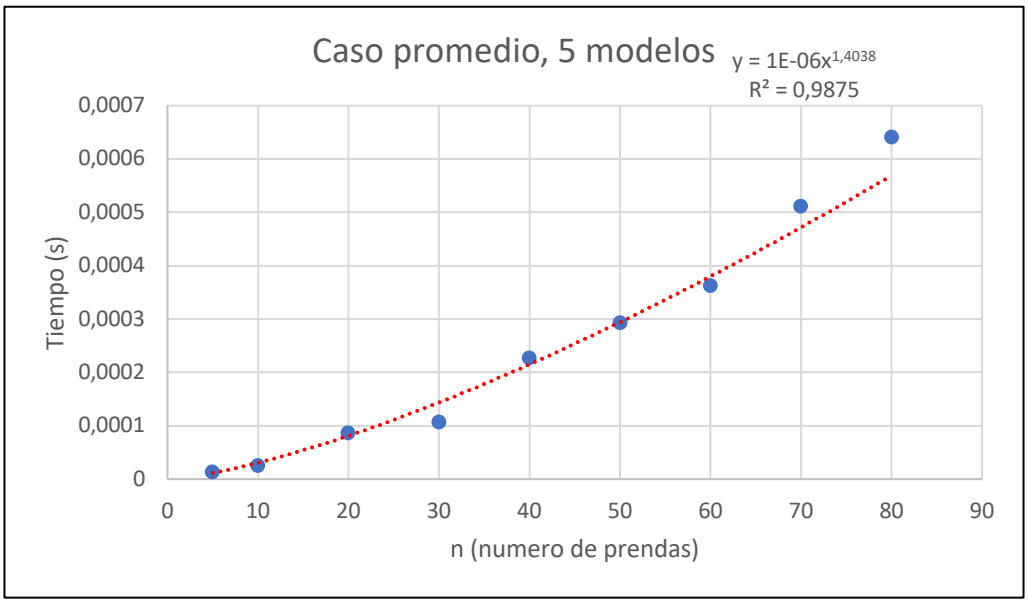

Figura 4. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso promedio y 5 modelos para diferentes valores del número de prendas. Fuente: propia.

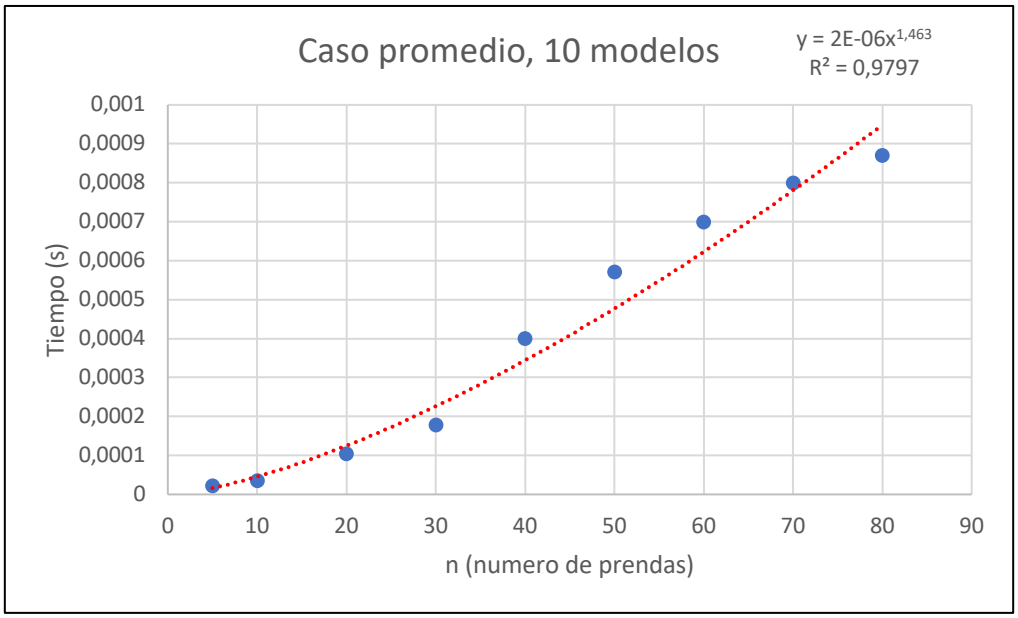

Figura 5. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso promedio y 10 modelos para diferentes valores del número de prendas. Fuente: propia.

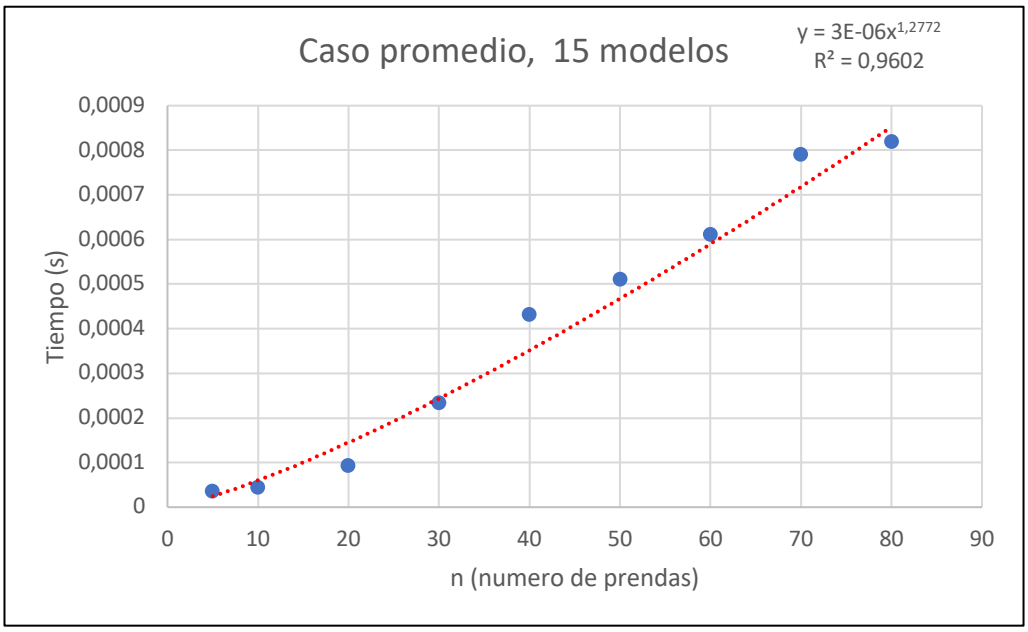

Figura 6. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso promedio y 15 modelos para diferentes valores del número de prendas. Fuente: propia.

Como se puede observar, se ha incluido una línea de tendencia de tipo potencial en la gráfica mostrada. Vemos cómo el valor de R^2, por lo general, es bastante alto, lo que nos parece indicar que en el caso promedio el tiempo crece casi linealmente conforme se aumenta el número de prendas (casi linealmente porque, como se puede ver en las ecuaciones de las líneas de tendencia de las gráficas, la potencia está cercana a 1). Hay que mencionar que debido al valor que se tomó para el dinero disponible en cada caso, casi siempre se obtenía la solución óptima (nótese que se generan muchos números y todos entre 1 y 200, es bastante probable que alguna combinación pueda sumar el valor que se tomó como dinero disponible), lo cual significa que se poda, no se sigue bajando el árbol cuando ya tenemos este valor. Por lo tanto, el tiempo obtenido dependerá de cómo de rápido se encuentre ese valor óptimo. Así, vimos que los datos tomados tenían mucha variabilidad, por lo que es normal que no haya una correspondencia tan clara como en el mejor caso.

También nos dimos cuenta de que, a partir del generador del caso promedio, podíamos generar un "pseudo-peor caso" (sería como el peor caso de todos los que podemos generar nosotros). Si tomamos como dinero disponible un valor muy grande, de manera que nunca pueda llegar a encontrar la solución óptima, nunca hay una serie de modelos de prendas que suman justo el dinero del que disponemos, se evitan muchas de las podas posibles, aunque no todas claro. Entonces, esto haría que se recorriesen muchos más nodos que antes, aunque no todos (no llega a ser el peor caso real, pero si es un caso peor que los promedios de forma general). Por lo tanto,

decidimos hacer las mismas pruebas que en el caso promedio, pero tomando 1000000 como dinero disponible, por ejemplo. Se obtuvieron las siguientes gráficas:

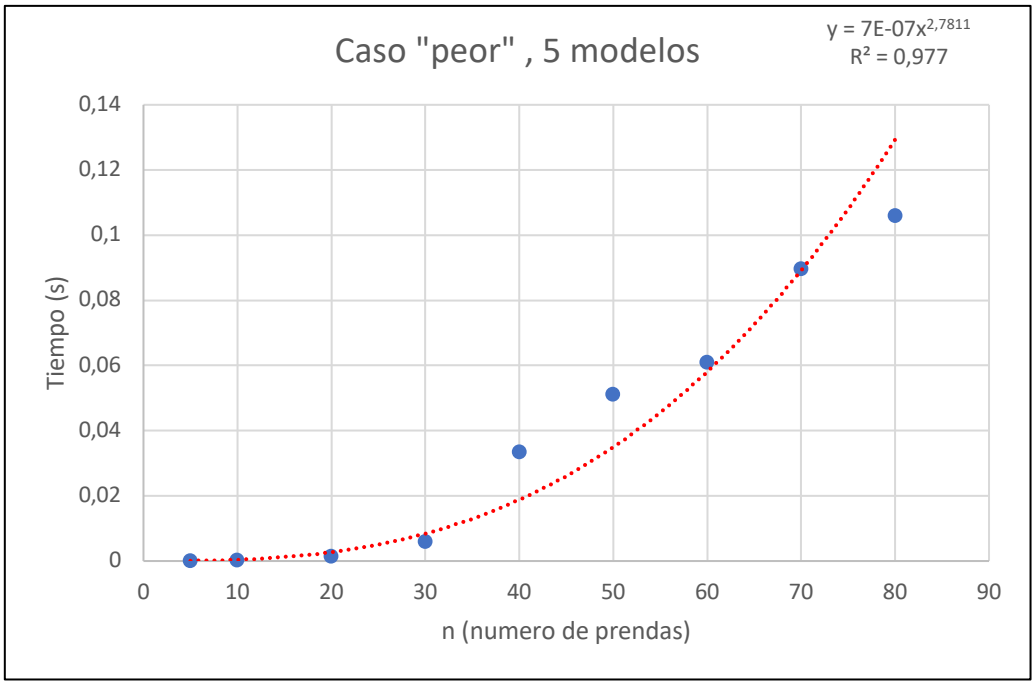

Figura 7. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso "peor" y 5 modelos para diferentes valores del número de prendas. Fuente: propia.

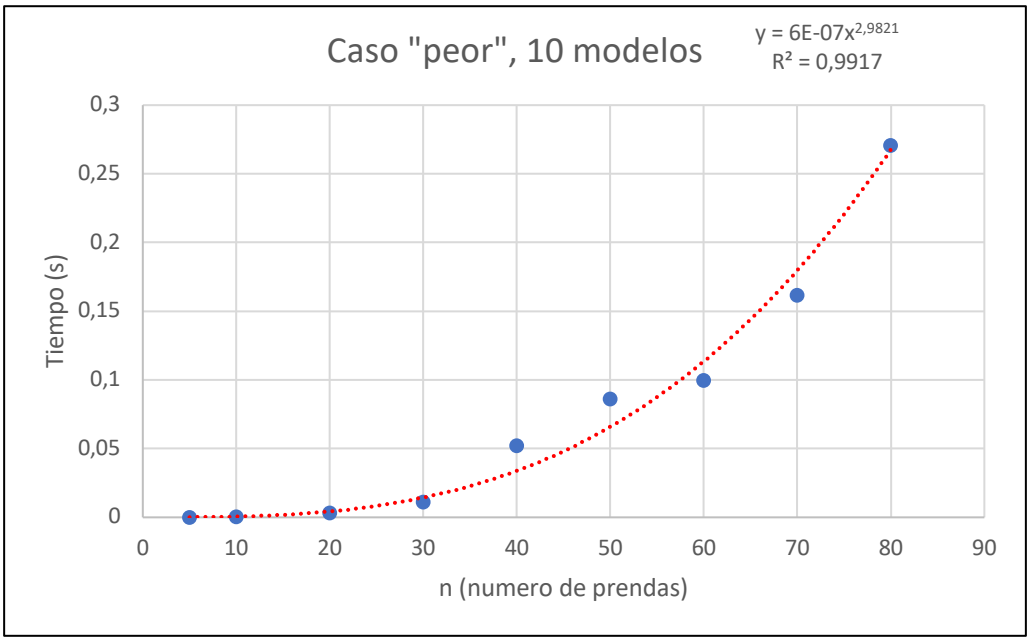

Figura 8. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso "peor" y 10 modelos para diferentes valores del número de prendas. Fuente: propia.

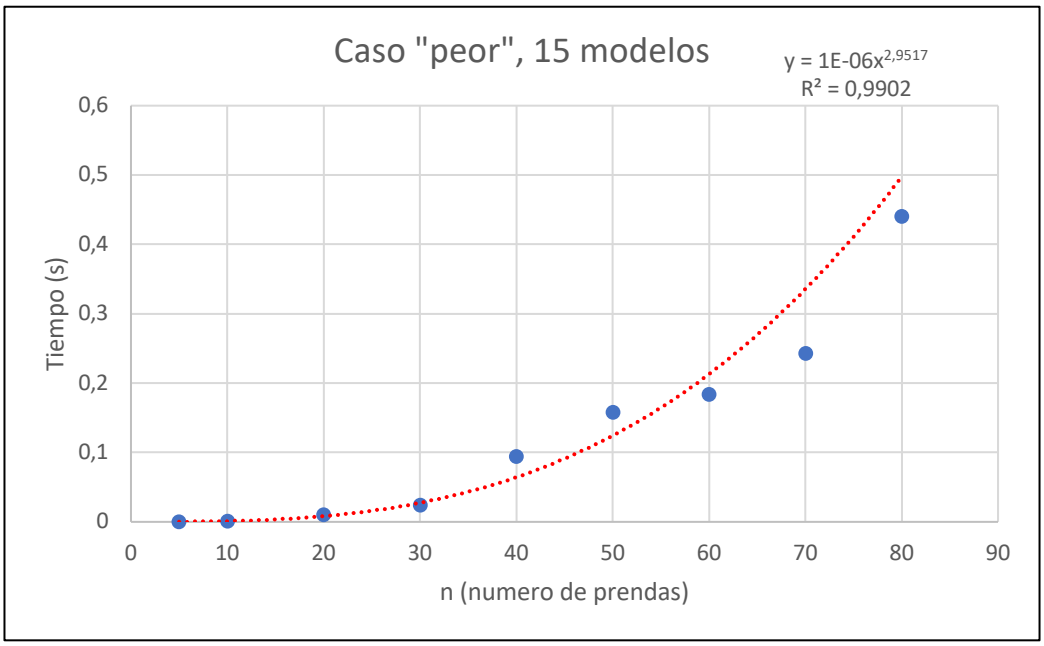

Figura 9. Gráfica para los tiempos de ejecución del algoritmo de backtracking en el caso "peor" y 15 modelos para diferentes valores del número de prendas. Fuente: propia.

Como se puede observar, en este "peor caso" que no tiene por qué recorrer todos los nodos, pero que recorre más que en el caso promedio (de forma general), se obtienen valores mucho más elevados para el tiempo de ejecución. Se ha incluido la línea de tendencia potencial, que era la que mejor se ajustaba a los datos, y la ecuación que sigue dicha línea de tendencia. Además, podemos observar que, claramente, conforme se aumenta el número de modelos por prenda, el tiempo de ejecución también aumenta.

Antes de pasar al siguiente apartado de esta memoria nos gustaría comentar cómo afecta el dinero al tiempo de ejecución. Claro está, el dinero no afecta directamente al orden del algoritmo, pero sí que afecta en cuántas podas se pueden llegar a hacer. Si el dinero disponible no es muy elevado, es más posible encontrar una solución que gaste todo el dinero y por lo tanto se harían bastantes podas. Sin embargo, como se ha podido comprobar, si la cantidad de dinero disponible es elevada, esto es más difícil (o incluso imposible si, como es el caso, se limita el precio de cada modelo), lo que provoca que el número de podas descienda y el tiempo de ejecución sea mayor, recorriéndose el árbol de forma más completa y, por tanto, acercándonos más al peor caso real del algoritmo.

4.5 CONTRASTE ENTRE ESTUDIO TEÓRICO Y EXPERIMENTAL

En el estudio teórico llegamos a la conclusión de que, en el mejor caso, nuestro algoritmo tendría un orden de ejecución lineal respecto al número de prendas. Si nos fijamos en la figura 3 de esta memoria, comprobamos que el estudio experimental llega a la misma conclusión, pues la línea de tendencia, que se acopla bastante bien según el valor de R^2 obtenido, sigue una ecuación lineal. Por lo tanto, creemos que queda bastante claro que en el mejor caso nuestro algoritmo tiene un crecimiento lineal conforme se aumenta el número de prendas.

En cuanto al peor caso, no pudimos detallar exactamente cuál es, pues no podemos controlar (o es muy difícil estudiarlas) qué o cuántas podas se llegan a producir. Por ello, lo que obtuvimos fue una cota superior del orden de complejidad, obtenida a partir del número total de nodos que podrían llegar a producirse en el árbol. En el estudio experimental, debido a esta dificultad para determinar el peor caso, no pudimos estudiarlo, pero sí que pudimos obtener datos para casos en los que se recorren y producen más nodos que en los casos promedio. En las figuras 7, 8 y 9 se puede observar cómo parece que el tiempo aumenta de forma potencial conforme se aumenta el valor del número de prendas, y que también aumenta el tiempo de ejecución conforme se aumenta el número de modelos de cada prenda. Esto concuerda con lo obtenido en el estudio teórico. Si nos fijamos en la expresión obtenida, vemos que el número de prendas no es lo único que influye, si no que también juegan un papel muy importante el número de modelos de cada prenda. Sin embargo, no sabemos el valor que puede tener la expresión obtenida en el estudio teórico, y por lo tanto no podemos comparar con mayor precisión ambos resultados. No obstante, si que parece claro que la expresión obtenida de forma teórica acota a los resultados obtenidos de forma experimental. A continuación, se muestra una figura que complementa al estudio experimental de este “peor” caso. En esta figura se puede ver más fácilmente cómo aumenta el tiempo conforme se aumenta el número de prendas y el número de modelos.

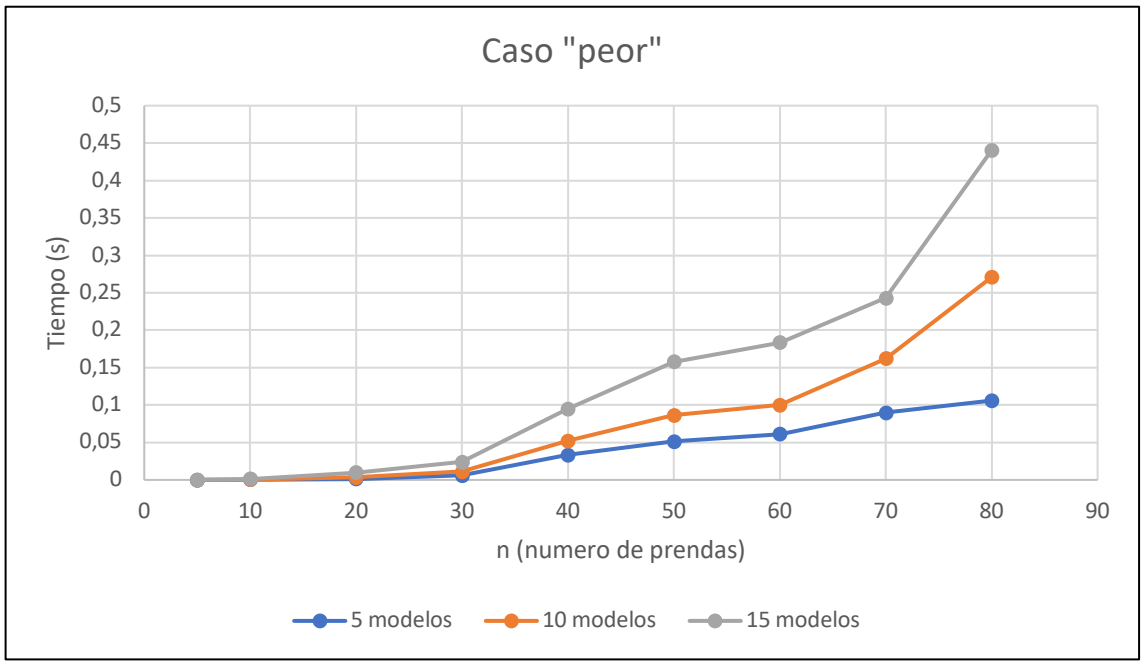

Figura 10. Tiempo en el "pseudo-peor caso" para 5, 10 y 15 modelos por prenda. Fuente: propia.

Por último, para el caso promedio, ya se ha comentado que los datos obtenidos tenían mucha variabilidad, debido a que el dinero disponible que se puso en cada caso permitía encontrar una solución que lo gastase todo y por tanto se hiciesen muchas podas (dependiendo de cómo de pronto se encontrase esta solución óptima el tiempo sería más alto o bajo, provocando una alta variabilidad en los datos). No obstante, también se puede intuir un crecimiento potencial (aunque la potencia parece bastante cercana a 1, casi sería crecimiento lineal). Además, es claro que estos casos promedios están acotados inferiormente por el orden de ejecución en el mejor caso, y superiormente por el orden en el peor caso. A continuación, se muestra una última gráfica que complementa al estudio experimental en el caso promedio:

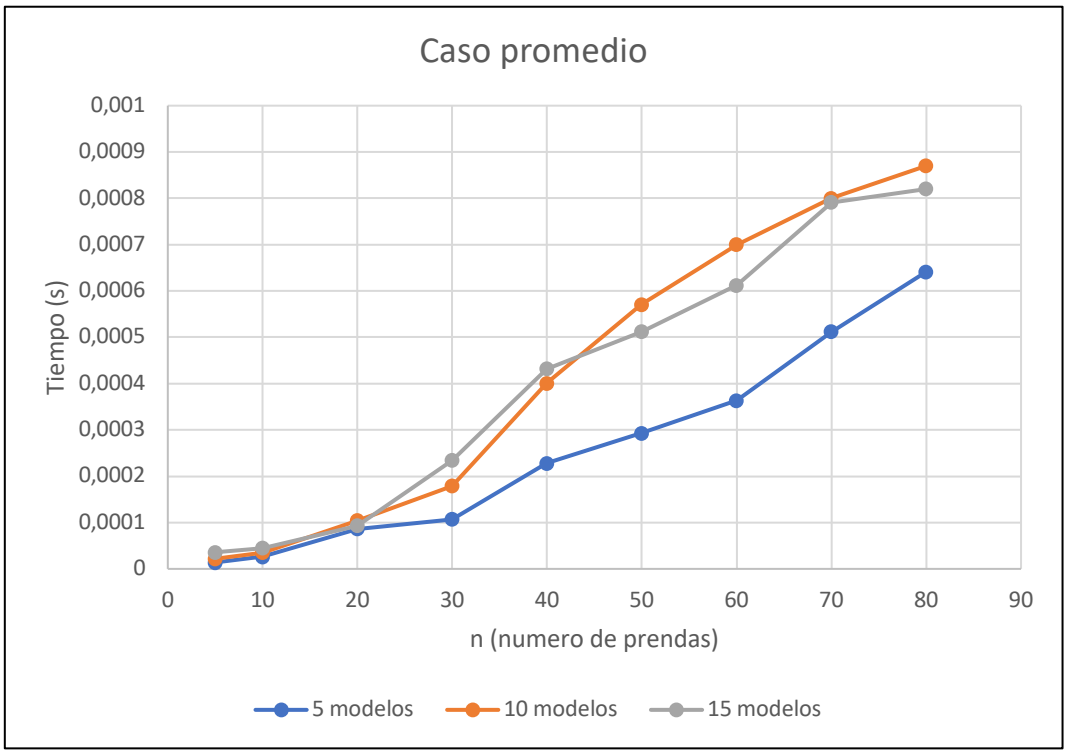

Figura 11. Tiempo en el caso promedio para 5, 10 y 15 modelos por prenda. Fuente: propia.

Como vemos parece que el tiempo aumenta, además de con el número de prendas, con el número de modelos, aunque con 15 modelos se llegaron a obtener resultados ligeramente inferiores que con 10. De nuevo, esto puede deberse a la alta variabilidad de los datos obtenidos, pero si que se puede intuir que el número de modelos también podría jugar un papel en el orden de ejecución en el caso promedio, al igual que lo hace con el peor caso. No obstante, no tenemos ningún resultado obtenido de forma teórica con el que hacer una mejor comparación.

5. CONCLUSIONES Y VALORACIONES PERSONALES

Como conclusión de esta práctica opinamos que finalmente, a pesar de ciertas dificultades encontradas, hemos podido completarla satisfactoriamente. Sin embargo, creemos que se podría mejorar el trabajo realizado en ciertos aspectos, como el estudio del teórico del tiempo de ejecución en el algoritmo de backtracking, pues su elevada dificultad conllevó que no se profundizase mucho en él. También creemos que, si se dispusiese de más tiempo, se podrían hacer más pruebas en el estudio experimental del backtracking que nos diesen una mayor cantidad de información, sobre todo en el caso promedio. También se podría haber probado el mejor caso de backtracking con diferentes números de modelos, para confirmar que el número de modelos no influye en este caso. Además, podríamos haber intentado realizar una poda mejor, pero de nuevo, esto hubiese llevado mucho tiempo y esfuerzo y no es seguro que la mejoría fuese notable.

Por otro lado, creemos que el trabajo realizado ha sido de gran utilidad para poder comprender y afianzar conocimientos de los temas 3 y 5 de la asignatura AED2, pues creemos que es una práctica profunda y que requiere una buena comprensión de la teoría aplicada para poder completarla de forma satisfactoria. También nos ha ayudado a ver cómo esta técnica puede ser de gran utilidad para la resolución de algunos problemas de programación.

Durante el desarrollo de esta práctica nos han ido surgiendo problemas y dudas, las cuales en la mayoría de los casos fueron resueltas con la ayuda del profesor Norberto Marín. Por ejemplo, en el algoritmo de backtracking estuvimos mucho tiempo atascados y gracias a sus indicaciones conseguimos hacer que el mooshak aceptase el programa. No obstante, en otros casos pudimos solventar el problema por nuestra cuenta tras dedicar un tiempo a reflexionar sobre él. Cabe destacar que en esta práctica en particular se hace uso del juez online, el mooshak, lo cual es de gran utilidad para comprobar el correcto funcionamiento de los algoritmos, pues también proporciona ejemplos extendidos de prueba muy útiles. También encontramos ayuda en los apuntes y en el libro de la asignatura, *Algoritmos y Estructuras de Datos Volumen 2*.

Para finalizar, tras sumar todo el tiempo dedicado a esta práctica y a la redacción de la presente memoria, creemos que aproximadamente dedicamos unas 25 horas a su realización.

6. REFERENCIAS BIBLIOGRÁFICAS

Giménez Cánovas, D.; Cervera López. J; García Mateos, G.; Marín Pérez, N. (2003) *Algoritmos y Estructuras de Datos – Volumen II – Algoritmos.*