UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 2: Divide y Vencerás

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

Autores y correos:
Jaime Corzo Galdó: jaimecrz04@correo.ugr.es
Marta Zhao Ladrón de Guevara Cano: mzladron72@correo.ugr.es
Sara Martín Rodríguez: smarrod02@correo.ugr.es

11 de abril de 2023

Índice

1. Autores

A continuación se va a detallar el trabajo de cada uno de los integrantes del grupo.

1. El código utilizado ha sido implementado entre los tres
2. El generador de casos - Marta
3. El análisis teórico, híbrido y empírico del caso base - Jaime y Sara
4. El análisis teórico del algoritmo DivideYVencerás - Sara
5. El análisis híbrido y empírico del algoritmo DivideYVencerás - Marta
6. Cálculo del valor del umbral teórico - Sara
7. Cálculo del valor del umbral óptimo - Marta
8. Umbrales de tanteo - Jaime
9. Cálculo gráfico del umbral - Jaime
10. Makefile - Sara
11. Memoria y presentación entre los tres

2. Objetivos

El objetivo de la práctica llevada a cabo ha sido estudiar la técnica de diseño de algoritmos "Divide y Vencerás"mediante la implementación de un algoritmo de este tipo para resolver un problema concreto que se describirá más adelante. Esto se ha hecho con el objeto de ver cémo este tipo de algoritmos mejoran la eficiencia y por tanto el tiempo de ejecución de códigos para resolver problemas, haciendo una buena elección del umbral y de la implementación. Todo esto, tanto el umbral como la implementación se detallarán a lo largo de la práctica.

3. Definición del problema

3.1. Descripción del problema

El problema a solucionar es: **Cálculo de las posiciones alcanzables**. Supongamos que tenemos un conjunto de posiciones del plano. Para cada posición conocemos sus coordenadas x e y; es decir, un punto p_i del mapa es de la forma $(p_i = (x_i, y_i))$. Se dice que un punto p_i alcanza a otro p_j si $x_i > x_j$ e $y_i > y_j$. Por simplicidad hemos supuesto que no existen dos puntos con las mismas coordenadas.

El algoritmo implementado tiene como objetivo devolver para cada punto el conjunto de posiciones que alcanza así como el número de posiciones alcanzables.

En este algoritmo se tiene cumple la propiedad transitiva; es decir si p_i alcanza a p_j y a su vez p_j alcanza a p_k, entonces se verifica que p_i alcanza a p_k. Esta propiedad tan inmediata es esencial y por tanto será utilizada en la implementación del algoritmo.

3.2. Descripción del entorno de análisis

A continuación, se detallan las características de los ordenadores de cada uno de los integrantes del grupo:

1. Jaime
    - Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
    - Memoria RAM de 16GB.
    - Capacidad de disco de 512.1GB.
    - Sistema operativo Ubuntu 22.04.2 LTS 64 bits.
    - Compilador g++
2. Marta
    - Procesador Intel Core i5-10210U CPU @ 1.60Hz x 8.
    - Memoria RAM de 8GB.
    - Capacidad de disco de 512.1GB.
    - Sistema operativo Ubuntu 20.04.5 LTS 64 bits.
    - Compilador g++
3. Sara
    - Procesador 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz x 8.
    - Memoria RAM de 8GB.
    - Capacidad de disco de 512 GB.
    - Sistema operativo Ubuntu 22.04.2 LTS 64 bits.
    - Compilador g++

3.3. Descripción del método de medición de tiempo

Para realizar la medición de los tiempos se ha hecho uso de la biblioteca de la STL, chrono, que permite calcular el tiempo de forma precisa. Para ello se deben incluir en el código las siguientes sentencias:

```
high_resolution_clock::time_point t_antes,t_despues;

duration<double> transcurrido;

t_antes = high_resolution_clock::now();

sort(posiciones.begin(),posiciones.end(), ordenar_x);
vector <Punto> sol;
sol=DivideYVenceras(posiciones);

t_despues = high_resolution_clock::now();

transcurrido=duration_cast<duration<double>>(t_despues - t_antes);
```

4. Consideraciones previas

Antes de entrar en detalle con los algoritmos implementados vamos a detallar una serie de datos y consideraciones que se han tenido en cuenta para ambos algoritmos.

Para guardar toda la información relativa a los puntos y los puntos a los que alcanza se ha hecho uso de un struct, que guarda la coordenada x, la coordenada y y una variable contador de las posiciones a las que alcanza cada punto. En él está definido un constructor que inicializa el punto con sus coordenadas x e y.

```
struct Punto{
    double x;
    double y;
    int pos_alcanzables;

    Punto(double i, double j){
        x=i;
        y=j;
        pos_alcanzables=0;
    }
};
```

Otro aspecto a tener en cuenta es que no puede haber dos puntos iguales; es decir con las mismas coordenadas, aunque si con una de ellas. Para controlar esto en la creación de números aleatorios, se ha utilizado inicialmente un set para ir almacenando los puntos, pues el set no permite repeticiones y para eso se ha utilizado un functor que compara cuando dos números son iguales, para luego volcar todos los puntos en el vector con el que se trabaja en el algoritmo. A continuación se muestra el código del generador de casos.

```
bool son_iguales(Punto P1, Punto P2){
    return (P1.x != P2.x || P1.y != P2.y);
}


int main(int argc, char* argv[]) {

    if (argc != 2){
        cerr << "Formato " << argv[0] << " <num_puntos>" << endl;
        return -1;
    }

    int n = atoi(argv[1]);

    if (n <= 0){
        cerr << "El numero de puntos ha de ser positivo" << endl;
        return -1;
    }

    ofstream fo;
    fo.open("ficheroDeSalida.txt");
    //Generamos los puntos aleatoriamente
    bool (*fn_pt)(Punto, Punto) =son_iguales;
    set<Punto, bool (*)(Punto, Punto)> posiciones(fn_pt);

    srandom(time(0));
    const int MIN = 0, MAX = 10;
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<float> distr(MIN, MAX);

    double i = 0;
    double j = 0;
    int k = 0;
    while (posiciones.size() < n) {
        i = distr(eng);
        j = distr(eng);
        Punto p(i, j);
        posiciones.insert(p);
        fo << p.x  << " ";
        fo << p.y << " ";
        ++k;

    }
    fo << endl;
    fo.close();

    return 0;
}
```

El generador de casos recibe como argumento el número de puntos que queremos que tenga nuestro vector y esos puntos van a ser almacenados en un fichero de texto, que servirá luego entrada para ejecutar los ejecutables del algoritmo específico y del Divide y vencerás. Para ello se van creando los números aleatoriamente y guardando en un set que no admite repeticiones. Aquí se hace uso del functor, que indica al set cuando dos puntos son iguales, pues este por defecto no lo sabe. De manera, que se crearán números aleatoriamente hasta que el set tengo tantos elementos como se ha indicado en el argumento. Y a la vez que los puntos son insertados en el set son envaidos al otro fichero, pues eso indica que no están repetidos.

5. Algoritmo específico

5.1. Diseño del algoritmo

El algoritmo específico se trata de un algoritmo iterativo que va a recorrer todos los puntos de plano que recibe el método y los va a ir comparando con el resto viendo a cuales alcanza. En el caso de que un punto p alcance a un punto q, p añadirá a su lista de puntos alcanzables a q y aumentará en uno la cantidad de puntos alcanzados. De manera que cuando acaben de ejecutarse los bucles cada punto tendrá almacenada una lista con todos los puntos a los que alcanza y una variable que indique la cantidad.

5.2. Detalles de implementación

La implementación del algoritmo es bastante sencilla e intuitiva, se basa en ir comparando todos los puntos entre si e ir comprobando cuáles se alcanzan y en tal caso aumentar la variable contadora. Luego es un doble bucle for anidado que en total recorren el vector n veces.

La función alcanza ha sido creada con el objeto de comprobar si un punto $p1$ alcanza a otro $p2$ en el caso de que las coordenadas de $p1$ sean estrictamente mayores que las coordenadas de $p2$.

```
bool alcanza(Punto p1,Punto p2){

    bool alcanza=false;
    if((p1.x>p2.x) && (p1.y>p2.y)){
        alcanza=true;
    }
    return(alcanza);
}

void especifico(vector<Punto> &a){

    //Recorremos todos los puntos y comprobando si se alcanzan
    for(int i= 0; i<a.size(); i++){
        for(int j=0; j<a.size(); j++){
            if(i!=j && alcanza(a[i],a[j])){
                a[i].posiciones_alcanzables.push_back(a[j]);
            }
        }
    }
}
```

5.3. Análisis de la eficiencia teórica, empírica e híbrida

5.3.1. Eficiencia teórica

Para estudiar la eficiencia teórica de este algoritmo tenemos que calcular el número de operaciones que se hacen dado un problema de tamaño n. Por un lado, vemos que todas las sentencias que hay dentro del bucle interno son de eficiencia $\mathcal{O}(1)$, pues añadir un elemento a un vector al final se hace en tiempo constante, supongamos que se ejecuta en un tiempo a. Además la función alcanza tiene también eficiencia $\mathcal{O}(1)$, luego el interior del bucle interno tiene eficiencia $\mathcal{O}(1)$. Por tanto, tenemos que estudiar la eficiencia de los dos bucles anidados, viendo cuantas iteraciones se hacen en cada uno. El bucle interior se ejecuta *a.size()* veces que es el número de puntos en el mapa; es decir, el tamaño del problema, luego el bucle interior realiza n iteraciones. Por otro lado, el bucle interno se ejecuta el número de veces indicado por el bucle exterior, en este caso n veces nuevamente. Luego, nos quedaría la siguiente fórmula:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a \Leftrightarrow \sum_{i=1}^{n}\sum_{j=1}^{n} a \Leftrightarrow a\sum_{i=1}^{n}\sum_{j=1}^{n} 1 \Leftrightarrow a\sum_{i=1}^{n} n = an^2$$

Como $an^2 \in \mathcal{O}(n^2)$, diremos que el algoritmo específico o caso base es de orden **cuadrático** o $\mathcal{O}(\mathrm{n}^2)$.

Contando el número de operaciones de la función se llega a que la función teórica que sigue este algoritmo es de la forma: $h(n) = an^2 + bn + c$, calculando estas

contantes ocultas, mediante un recuento de las operaciones ejecutadas en la función previamente mostrada, tenemos que $h(n) = 18n^2 + 6n + 3$,

5.3.2. Eficiencia empírica

Para estudiar la eficiencia empírica de este algoritmo, hemos medido los recursos empleados (tiempo) para distintos tamaño de las entradas., en este caso, el número de puntos. Como indicamos antes, para medir los tiempos hemos usado la biblioteca de la STL, chrono. Así hemos obtenido los tiempos de ejecución para los distintos tamaños de entrada.

Num. puntos	**Tiempo (seg)**
5000	0.118866
10000	0.403069
15000	0.928421
20000	1.687640
25000	2.660610
30000	3.845240
35000	4.921260
40000	6.519530
45000	8.243520
50000	10.204000
55000	12.368100
60000	14.848300
65000	18.001500
70000	21.414800
75000	24.900600
80000	28.618500
85000	32.640000
90000	35.525000
95000	38.336700
100000	42.437600
105000	46.271900
110000	53.729000
115000	58.253400
120000	63.308700
125000	82.872300

Cuadro 1: Tiempos del algoritmo base hasta 125000 puntos con saltos de 5000

Num. puntos	**Tiempo (seg)**
400	0.002090
500	0.003199
600	0.004695
700	0.006458
800	0.008338
900	0.010500
1000	0.008314
1100	0.007294
1200	0.007242
1300	0.008411
1400	0.009938
1500	0.011691
1600	0.012831
1700	0.014547
1800	0.016163
1900	0.017811
2000	0.018893
2100	0.021964
2200	0.024087
2300	0.026437
2400	0.029046
2500	0.031169
2600	0.033466
2700	0.035759
2800	0.038650
2900	0.039111
3000	0.038587
3100	0.041709
3200	0.044334
3300	0.046329
3400	0.050241
3500	0.053611
3600	0.056606
3700	0.060174
3800	0.063244
3900	0.065574
4000	0.070508
4100	0.074268
4200	0.076419
4300	0.091950
4400	0.090480
4500	0.087188

Cuadro 2: Tabla del algoritmo base hasta 4500 puntos con saltos de 100

5.3.3. Eficiencia híbrida

El cálculo teórico realizado previamente nos da la expresión general, pero asociada a cada término de esta expresión aparece una constante de valor desconocido. Luego necesitamos obtener el valor de esas constantes. La forma de averiguar estos valores es ajustar la función a un conjunto de puntos. En nuestro caso, la función es la que resulta del cálculo teórico, el conjunto de puntos lo forman los resultados del análisis empírico y para el ajuste emplearemos regresión por mínimos cuadrados.

Como hemos visto el algoritmo es cuadrático luego usamos una función polinómica de segundo grado. $T(n) = a_2 * n^2 + a_1 * n + a_0$ Obtenemos las constantes ocultas, y así podemos saber el tiempo para cualquier tamaño de entrada n. Al realizar el ajuste se obtienen:

- Para 5000 puntos, la ecuación $t(n) = 4{,}11143 \cdot 10^{-9} \cdot n^2 + 7{,}82516 \cdot 10^{-7} \cdot n + 0{,}00187568$ con un coeficiente de determinación de $5{,}72535 \cdot 10^{-6}$
- Para 125000 puntos, la ecuación $t(n) = 5{,}48833 \cdot 10^{-9} \cdot n^2 - 0{,}000117231 \cdot n + 1{,}81929$ con un coeficiente de determinación de $6{,}104 \cdot 10^{-5}$

Como el residuo es casi cero vemos que el ajuste es muy bueno.

Ahora vemos como ajusta esta función a nuestros datos:

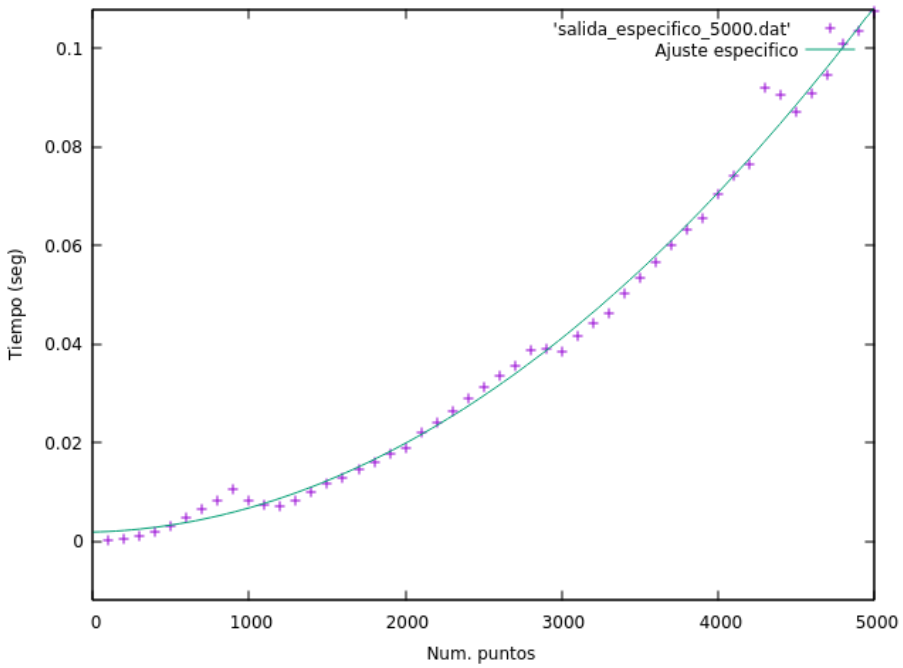

Figura 1: Gráfica del análisis empírico del algoritmo específico para 5000 puntos

Ahora podemos estimar el tiempo de ejecución del algoritmo para un tamaño de puntos n.

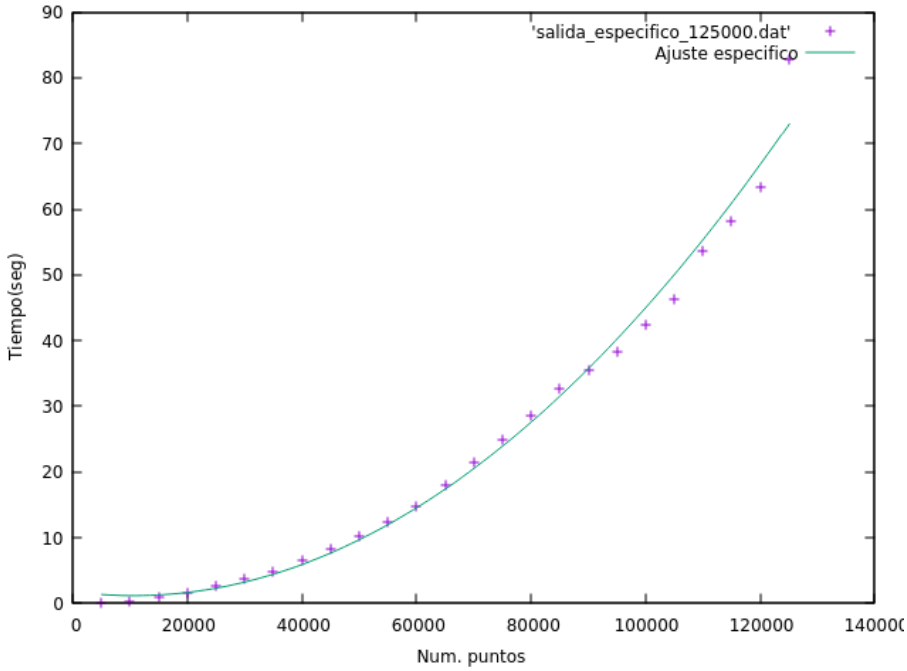

Figura 2: Gráfica del análisis empírico del algoritmo específico para 125000 puntos

6. Algoritmo divide y vencerás

6.1. Diseño del algoritmo

El algoritmo divide y vencerás se basa en la descomposición de un problema en subproblemas de tamaño menor aplicando la recursividad. Para ello se debe especificar una función de parada, por debajo de la cual ya no se aplique la recursividad sino un algoritmo específico (no recursivo) que va a resolver problemas de tamaño pequeño, para se hará uso del algoritmo específico previamente desarrollado.

A lo largo de esta sección se va realizar un estudio sobre el algoritmo divide y vencerás puro; es decir, donde la la condición de parada se basa en llegar a un vector con únicamente un punto, pues en este caso el problema a resolver es un vector de tamaño, que se irá subdividiendo en vectores mas pequeños. A continuación, se estudiará el valor del umbral; es decir, el punto en el que se dejará de usar la recursividad para usar el método iterativo.

6.2. Detalles de implementación

A continuación, se muestra el código del divide y vencerás puro. Se divide entres funciones que explicadas de abajo a arriba son:

1. ***DivideYVencerás(vector Punto v)*** : esta función recibe el vector que contiene todos los puntos del problema, los cuales han sido creados previamente de forma aleatoria, y esta llamará a otra función con el mismo nombre pero con otros parámetros, que será la función que se llamará a si misma de manera recursiva.

2. ***DivideYVenceras (vector Punto v, int inf, int sup))*** : el vector v que esta función recibe va a ser siempre el mismo, la diefrencia radica en los otros dos parámetros que vana ir indicando sobre que posiciones del vector estamos trabajando, y a partir de ellas se hará la nueva partición del vector.

3. ***Fusion (vector Punto der, vector Punto izd)*** : este método se encargará de unir los dos vectores resultantes de llamar a la función anterior.

```
vector<Punto>Fusion(const vector<Punto> &izq,const vector<Punto> &dcha){

    vector<Punto> solucion;
    int tam_izq = izq.size();
    int tam_dcha = dcha.size();

    double max_izq = max_element(izq.begin(),izq.end(), ordenar_x)->x;
    int i = 0; //posicion vector izquierda
    int d = 0; //posicion vector derecha
    int alcanzables = 0;
    int alcanzables_estricto= 0;
    Punto pto_dcha = dcha.at(d);
    Punto pto_izq = izq.at(i);

    do{
        // Tomamos un punto de cada vector
        if (d < tam_dcha) {
            pto_dcha = dcha.at(d);
        }
        if (i < tam_izq) {
            pto_izq = izq.at(i);
        }
        bool en_lim = (d == tam_dcha || i == tam_izq);

        if (en_lim) {
            if (d == tam_dcha) {
                while (i < tam_izq) {
                    solucion.push_back(izq.at(i));
                    i++;
                }
            }
        }
```

Continuación de la función fusión

```
            else {
                if (i == tam_izq) {
                    while (d < tam_dcha) {
                        solucion.push_back(dcha.at(d));
                        if (dcha.at(d).x == max_izq) {
                            solucion.back().pos_alcanzables
                                        += alcanzables_estricto;
                        } else {
                            solucion.back().pos_alcanzables
                                                += alcanzables;
                        }
                        d++;
                    }
                }
            }
        }
        else {
            if ((pto_izq.y < pto_dcha.y)) {
                solucion.push_back(pto_izq);
                alcanzables++;
                if (pto_izq.x != max_izq) {
                    alcanzables_estricto++;
                }
                i++;
            }
            else{
                solucion.push_back(pto_dcha);
                if (pto_dcha.x == max_izq) {
                    solucion.back().pos_alcanzables
                                += alcanzables_estricto;
                } else {
                    solucion.back().pos_alcanzables
                                        += alcanzables;
                }
                d++;
            }
        }
    }while(d<tam_dcha||i<tam_izq);

    return solucion;

}
```

Implementación divide y vencerás

```
vector<Punto> DivideYVenceras (const vector<Punto> &v,int inf, int sup ){

    vector<Punto> solucion;

    int tam_v=sup-inf;

    // condicion parada: el vector tiene un tamaño menor o igual que el umbral
    if (tam_v > UMBRAL) {
        // llamada recursiva con los dos nuevos vectores
        vector<Punto> izq=DivideYVenceras(v, inf, ceil((double)(inf+sup)/2));
        vector<Punto> dcha=DivideYVenceras(v,ceil((double) (inf+sup)/2) , sup);

        // unir los vectores
        solucion = Fusion(izq,dcha);

    }
    else{
        vector<Punto> aux;
        for(int i = inf; i< sup; i++) aux.push_back(v.at(i));
        solucion = dyv_basico(aux);
        sort(solucion.begin(), solucion.end(), ordenar_y);
    }

    return solucion;
}

vector<Punto> DivideYVenceras (const vector<Punto> &v){

    return (DivideYVenceras(v,0,v.size()));
}
```

Como premisa para que el algoritmo divide y vencerás supone el vector que se mande a la función DivideYVencerás debe de estar ordenado de menor a mayor por las abscisas de los puntos, operación que se hace en el *main* antes de llamar a dicha función y cuya parte del código se muestra a continuación, también con el objetivo de mostrar la creación aleatorio de los puntos.

```
int main(int argc, char* argv[]) {


    if (argc != 2){
        cerr << "Formato " << argv[0] << " <fichero_puntos>" << endl;
        return -1;
    }

    // Nombre del fichero donde están los datos de entrada
    char *nom_fich= argv[1];

    ifstream fin;
    fin.open(nom_fich);
    int num_puntos=0;
    vector<Punto> puntos;

    // Lectura de los datos y se guardan en un  vector
   if(fin){
        while(fin){
            double i,j;
            fin >> i;
            fin >> j;
            if(fin){
                Punto p(i,j);
                cout << "Punto (" << p.x <<  "," << p.y << ")" << endl;
                puntos.push_back(p);
                num_puntos++;
            }
        }

        fin.close();
    }
    else{
        cout << "ERROR: no se ha podido abrir el fichero"
        << "que contiene los puntos." << endl;
    }

    high_resolution_clock::time_point t_antes,t_despues;
    duration<double> transcurrido;
    t_antes = high_resolution_clock::now();

    // Se ordenan los vectores por abscisas de menor a mayor
    sort(puntos.begin(),puntos.end(), ordenar_x);

    vector <Punto> sol;
    sol=DivideYVenceras(puntos); // Se llama al algoritmo

    t_despues = high_resolution_clock::now();

    transcurrido = duration_cast<duration<double>> (t_despues - t_antes);
```

```
    cout << num_puntos << "  " << transcurrido.count() << endl;

        // Los volvemos a ordenar para mostrar los resultados
        sort(sol.begin(),sol.end(), ordenar_x);
        for(int i=0; i<sol.size();i++){
            cout << "Punto (" << sol[i].x <<  "," << sol[i].y << "):"
                 << sol[i].pos_alcanzables << endl;
        }


        return 0;
    }
```

Los ejecutables reciben como parámetro el nombre del fichero en el que se encuentran los puntos, luego en el main hace la lectura de esos datos y los va guardando en el vector de puntos, para a continuación ordenarlos de menor a mayor por las abscisas y ejecutar el algoritmo Divide y Vencerás.

6.3. Análisis de la eficiencia teórica, empírica e híbrida

6.3.1. Eficiencia teórica

El cálculo de la eficiencia teórica se basa en calcular el número de operaciones que se van a realizar para solucionar el problema en función del tamaño del problema, en este caso, el tamaño del problema va a ser el tamaño del vector, que vamos a suponer que es de tamaño n. La eficiencia de este algoritmo va a ser el máximo de: la eficiencia de ordenar el vector y la eficiencia de la función divide y vencerás:

$$T(n) = max\{\mathcal{O}(sort),\ \mathcal{O}(DivideYVenceras)\}$$

La eficiencia del algoritmo sort de la STL es $\mathcal{O}(nlogn)$ pues utiliza el método de ordenación de quicksort, y como se vio y estudio en la Práctica 1 se trata de un algoritmo con eficiencia $nlog(n)$.

Por otro lado, la eficiencia teórica de la función divide y vencerás es también $\mathcal{O}(nlog(n))$, como se va a probar a continuación. En la función se hacen dos llamadas recursivas a si misma pero con un problema de tamaño menor, concretamente, el tamaño del problema se reduce a la mitad, a $n/2$, a lo que habría que sumar la eficiencia que suponga la función de fusión. Si nos fijamos en ella, previamente implementada, veremos que el while principal tiene como objetivo recorrer los vectores izquierda y derecha por completo lo que suponen a los sumo n iteraciones, con los while internos se avanzan los índices que recorren estos vectores, luego en el peor caso la eficiencia de la función de fusión es lineal $\mathcal{O}(n)$. Por tanto, la eficiencia de la función DivideYVencerás es: $T(n) = 2T(n/2) + n$.

Resolución de la ecuación de recurrencia:
Cambio de variable: $n = 2^m \Rightarrow m = log_2(n)$
$T(2^m) = 2 * T(2^{m-1}) + 2^m \Rightarrow T(2^m) - 2 * T(2^{m-1}) = 2^m$

Parte homogénea:
$T(2^m) - 2 * T(2^{m-1}) = 0 \Rightarrow p_H(x) = x - 2$

Parte no homogénea:
$2^m = b_1^m * q_1(m)$ luego $b_1 = 2$, $q_1(m) = 1$ con grado $d_1 = 0$
$p(x) = (x-2)(x-b_1)^{d_1+1} = (x-2)^2$ La raíz es 2 con multiplicidad 2

$t_m = c_{10}2^m * m^0 + c_{11} * m^1 * 2^m$

Deshacemos el cambio:
$t_n = c_{10} * n + c_{11} * n * log(n)^1 \ \in \ \mathcal{O}(nlog(n))$

Llegamos a la conclusión de que se trata de la función tiene una eficiencia $\mathcal{O}(nlog(n))$.

Luego la eficiencia del algoritmo es:

$$T(n) \ = \ \mathcal{O}(nlog(n)) \ + \ \mathcal{O}(nlog(n)) \ = \ max\{\mathcal{O}(nlog(n)), \mathcal{O}(nlog(n))\} \ \in \ \mathcal{O}(nlog(n))$$

Concretamente el algoritmo divide y vencerás sigue una función de la siguiente forma: $t(n) = 2T(\frac{n}{2}) + g(n)$, donde $g(n) = dn + e$ es la eficiencia de la función de fusión, la función teórica de la función de combinación es, cotando el número de operaciones que se realizan $g(n) = 20n + 31$

6.3.2. Eficiencia empírica

De la misma forma, para estudiar la eficiencia empírica del algoritmo en su versión Divide y Vencerás, se ha medido el tiempo de ejecución para distintos tamaños de entrada que en este caso han sido el número de puntos a analizar. Los resultados han sido los siguientes:

Tamaño	**Tiempo (seg)**
50000	0.025156
100000	0.051948
150000	0.072745
200000	0.086813
250000	0.106214
300000	0.143723
350000	0.160916
400000	0.184779
450000	0.210808
500000	0.238141
550000	0.312173
600000	0.310401
650000	0.313213
700000	0.424194
750000	0.384678
800000	0.397102
850000	0.425577
900000	0.425528
950000	0.501721
1000000	0.511851
1050000	0.568485
1100000	0.575554
1150000	0.593837
1200000	0.640391
1250000	0.640064

Cuadro 3: Tabla del algoritmo dyv hasta 1250000 puntos con saltos de 150000

Num. puntos	Tiempo (seg)
200	0.000272
400	0.000509
600	0.000855
800	0.001113
1000	0.001540
1200	0.000933
1400	0.000716
1600	0.000704
1800	0.000671
2000	0.000730
2200	0.000881
2400	0.000917
2600	0.000983
2800	0.001065
3000	0.001135
3200	0.001212
3400	0.001276
3600	0.001347
3800	0.001426
4000	0.001487
4200	0.001671
4400	0.001757

Cuadro 4: Tabla del algoritmo base hasta 4400 puntos con saltos de 200

6.3.3. Eficiencia híbrida

Eficiencia híbrida de la función fusión

Como se puede ver a simple vista la función que ajusta a los datos obtenidos de ejecutar únicamente la función fusión es una función lineal. Realizando un ajuste por mínimos cuadrados obtenemos que la ecuación $f(x) = 3{,}15148 * 10^{-7}x + 0{,}000133813$ se ajusta con un coeficiente de determinación igual a $5{,}79917 * 10^{-7}$; es decir, prácticamente nulo, lo que se traduce en que el ajuste es muy bueno.

Eficiencia híbrida de la función recursiva Divide y Vencerás

Realizando de nuevo el ajuste por mínimos cuadrados obtenemos:

- Para 5000 puntos, la ecuación $f(x) = 4{,}96651 \cdot 10^{-8} \cdot x \cdot \log x$ con un coeficiente de determinación de $1{,}37835 \cdot 10^{-7}$
- Para 12500000 puntos, la ecuación $f(x) = 3{,}75939 \cdot 10^{-8} \cdot x \cdot \log x$ con un coeficiente de determinación de 0.000423845.

Como el residuo es casi cero vemos que el ajuste es muy correcto.

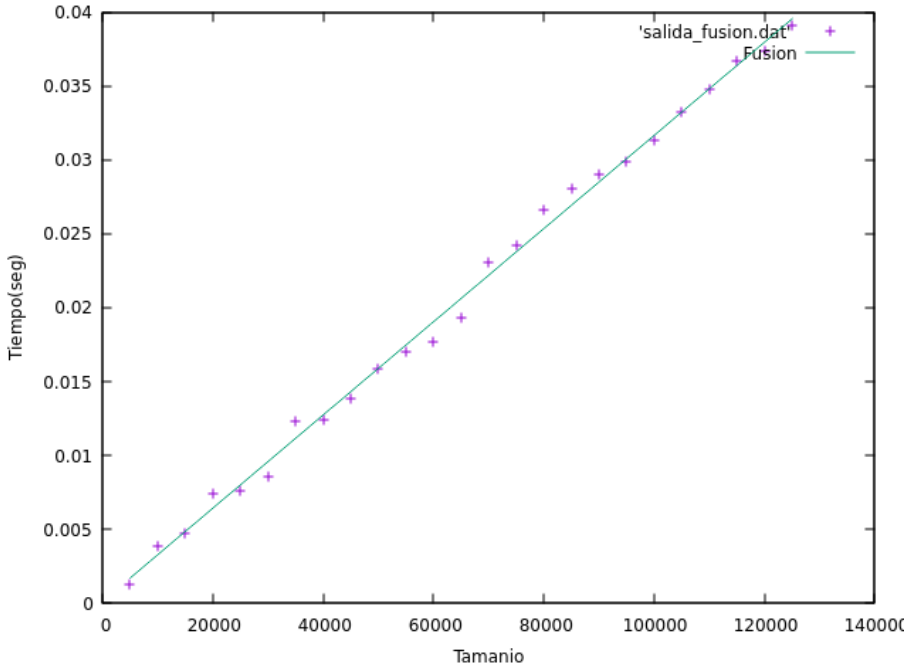

Figura 3: Gráfico función fusión

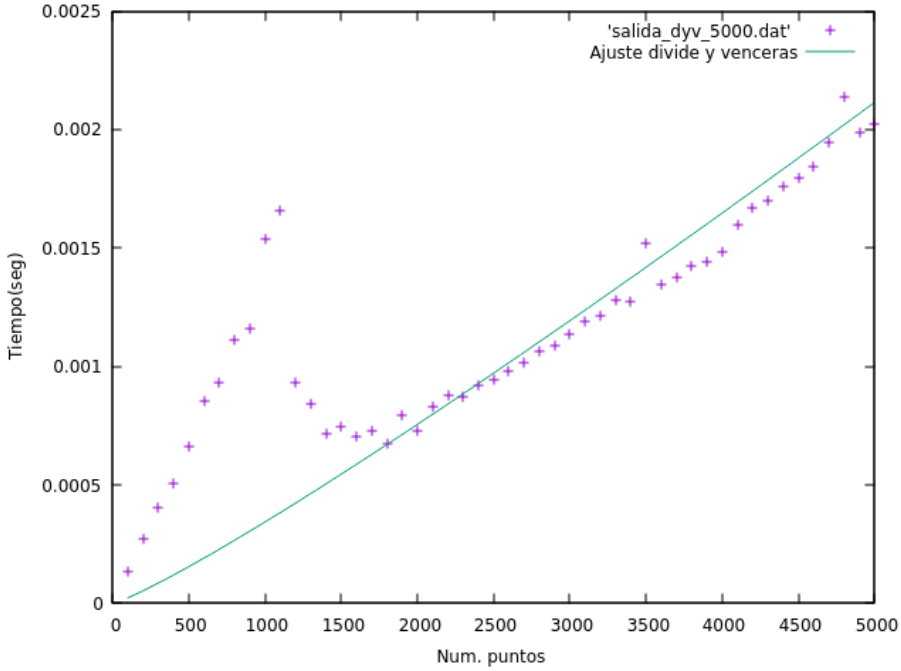

Figura 4: Gráfico función Divide y Vencerás para 5000 puntos

6.4. Cálculo de umbrales teórico, óptimo y de tanteo

En la siguiente sección se va a hacer un análisis del umbral; es decir, el punto en el que se pasa de usar la recursividad a usar un método iterativo para tamaños de problema pequeños.

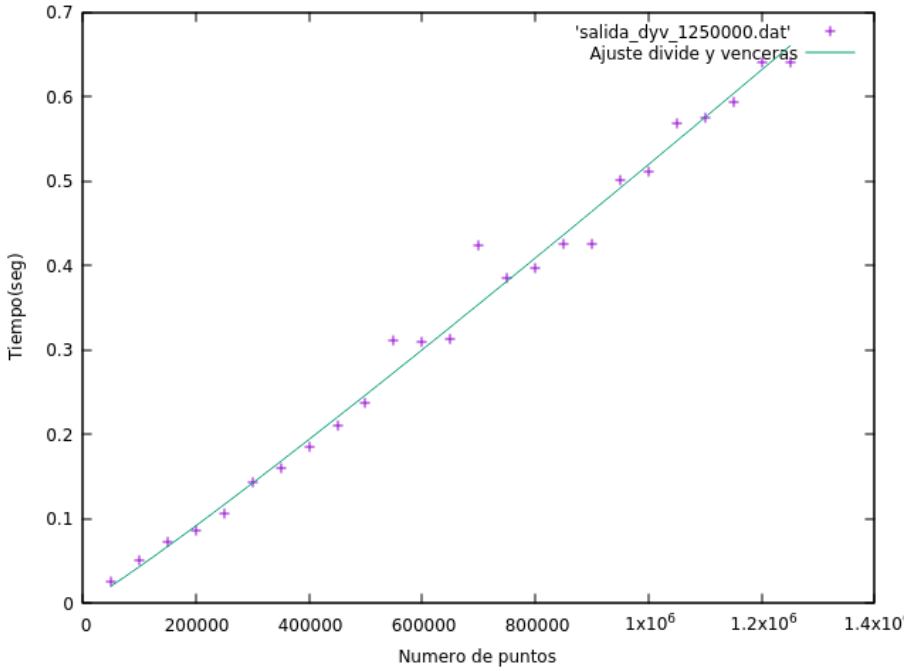

Figura 5: Gráfico función Divide y Vencerás para 125000 puntos

6.4.1. Umbral teórico

El umbral teórico es un valor indicativo, pues a pesar de no ser un valor único si que lo es para cada implementación; es decir, depende de factores tales como el algoritmo base, la implementación, los casos estudiados... El cálculo del valor del umbral teórico compara el tiempo del algoritmo básico con el del divide y vencerás usando sólo un nivel de recursividad. Para ello debemos de partir de las expresiones del tiempo de ejecución del algoritmo base, la expresión teórica, y la expresión recurrente del tiempo de ejecución para el método recursivo. Se basa en igualar ambas expresiones y calcular el tamaño del problema para el que ambos tiempos son iguales. Se busca el punto de corte de ambas funciones, n_o

$$t(n) = \begin{cases} h(n) & si\ n \leq n_o \\ 2t(\frac{n}{2}) + g(n) & si\ n > n_o \end{cases}$$

Particularizándola a nuestro problema tenemos:

$$t(n) = \begin{cases} an^2 + bn + c & si\ n \leq n_o \\ 2t(\frac{n}{2}) + dn + e & si\ n > n_o \end{cases}$$

Igualando ambas expresiones tenemos que:

$$h(n) = t(n);\ an^2 + bn + c = 2h(\tfrac{n}{2}) + g(n);\ an^2 = 2(\tfrac{n}{2})^2 + 2b\tfrac{n}{2} + 2c + dn + e;$$

$$\tfrac{an^2}{2} - dn - (c+e) = 0;\ n = \tfrac{d \pm \sqrt{d^2 + 2*a*c + 2*a*e}}{a}$$

Si las funciones obtenidas de un análisis teórico de ambos métodos son:

$h(n) = an^2 + bn + c;\ h(n) = 18n^2 + 6n + 3$ y

$g(n) = dn + e; g(n) = 20n + 31$

Sustituyendo en la ecuación nos queda que $n_o = 3{,}34$, luego el umbral teórico es 3.

6.4.2. Umbral óptimo

Para hallar el umbral óptimo se va a utilizar el método híbrido, ya que con el método teórico se ha estimado el umbral con el número de operaciones obviando el tiempo real que tardan los algoritmos en ejecutarse. Anteriormente, se han obtenido las funciones que modelan los tiempos de ejecución de dichos algoritmos a través de una estimación por mínimos cuadrados de los resultados obtenidos para 5000 puntos:

- $h(n) = 4{,}96651 \cdot 10^{-8} \cdot n \cdot \log n$, para el algoritmo Divide y Vencerás
- $g(x) = 4{,}11143 \cdot 10^{-9} \cdot n^2 + 7{,}82516 \cdot 10^{-7} \cdot n + 0{,}00187568$, para el algoritmo recursivo.

Tal y como se aprecia en la siguiente imagen, ambas funciones tienen gráficas que se aproximan cerca del 0 pero que no llegan a intersecarse.

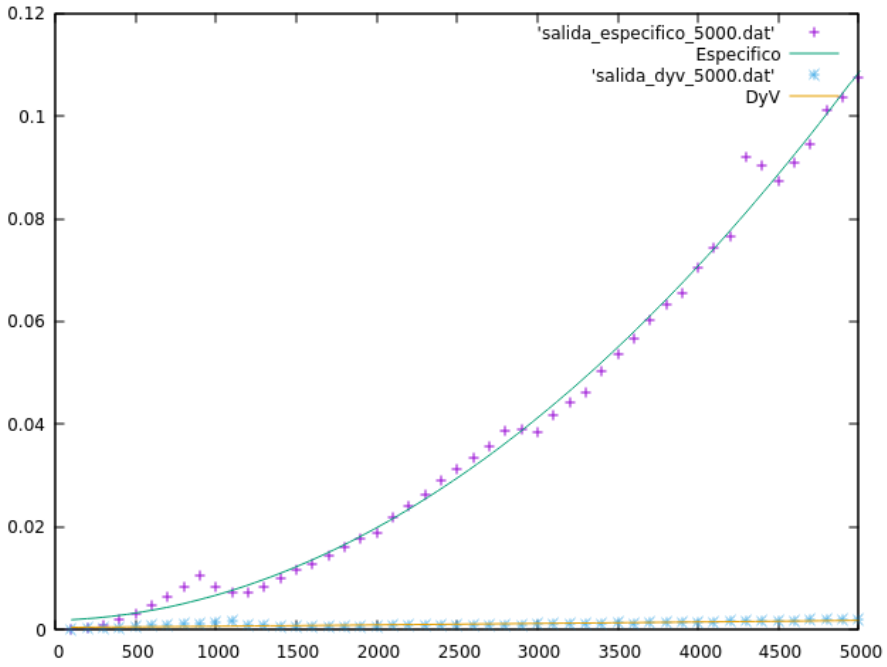

Figura 6: Ajuste de algoritmos específico y Divide y Vencerás

Luego el algoritmo Divide y Vencerás es siempre la mejor opción para obtener la solución a este problema. El umbral óptimo es por tanto n=0.

6.4.3. Umbral de tanteo

Para obtener el umbral óptimo se han testeado varios valores alrededor del umbral óptimo obtenido que ha sido $n_0 = 0$. Dichos valores son los umbrales de tanteo

y han sido los siguientes. Los valores elegidos para estudiar el umbral de tanteo dado que el umbral óptimo es 0, van a ser números superiores, y vamos a probar tanto números cercanos al umbral óptimo y teórico, como algunos mayores para demostrar que verdaderamnete el algoritmo divide y vencerás presenta tiempos mejores.

Num. puntos	Tiempo (seg)
50000	0.0174737
100000	0.0314006
150000	0.0516407
200000	0.0640079
250000	0.079789
300000	0.107961
350000	0.11832
400000	0.138087
450000	0.14871
500000	0.1696
550000	0.210206
600000	0.232813
650000	0.240078
700000	0.251213
750000	0.267672
800000	0.279598
850000	0.297719
900000	0.322246
950000	0.329812
1000000	0.349235
1050000	0.40169
1100000	0.418631
1150000	0.444351
1200000	0.457781
1250000	0.473763

Cuadro 5: Umbral=3

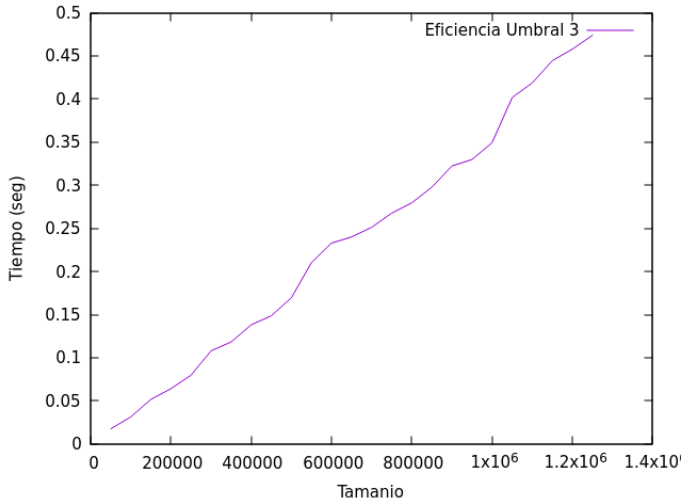

Figura 7: Grafica dyv usando UMBRAL=3

Num. puntos	Tiempo (seg)
50000	0.0164344
100000	0.032295
150000	0.049365
200000	0.0655035
250000	0.0786015
300000	0.104486
350000	0.126743
400000	0.147389
450000	0.163206
500000	0.173078
550000	0.21152
600000	0.22559
650000	0.244911
700000	0.267823
750000	0.283538
800000	0.300111
850000	0.31657
900000	0.327995
950000	0.352982
1000000	0.370641
1050000	0.412154
1100000	0.445645
1150000	0.453466
1200000	0.483684
1250000	0.505577

Cuadro 6: Umbral=10

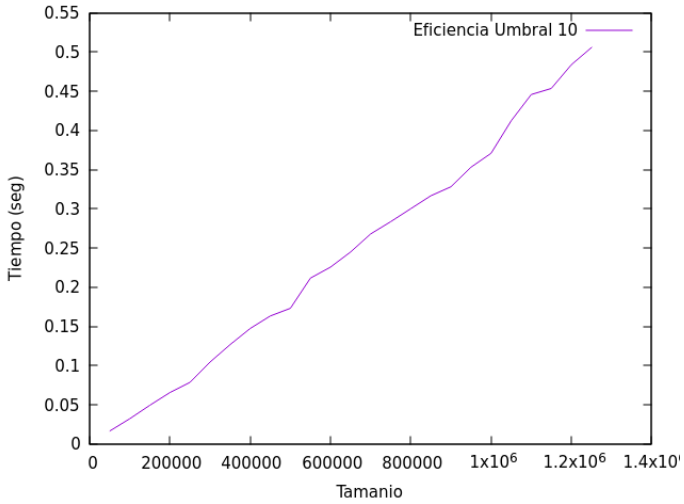

Figura 8: Grafica dyv usando UMBRAL=10

Num. puntos	Tiempo (seg)
50000	0.0160323
100000	0.0293798
150000	0.043159
200000	0.0585241
250000	0.0667228
300000	0.0896558
350000	0.104427
400000	0.124325
450000	0.127891
500000	0.138373
550000	0.177192
600000	0.192085
650000	0.205856
700000	0.224015
750000	0.240145
800000	0.254294
850000	0.245732
900000	0.273749
950000	0.288256
1000000	0.307119
1050000	0.348086
1100000	0.402966
1150000	0.610065
1200000	0.68566
1250000	0.698817

Cuadro 7: Umbral = 50

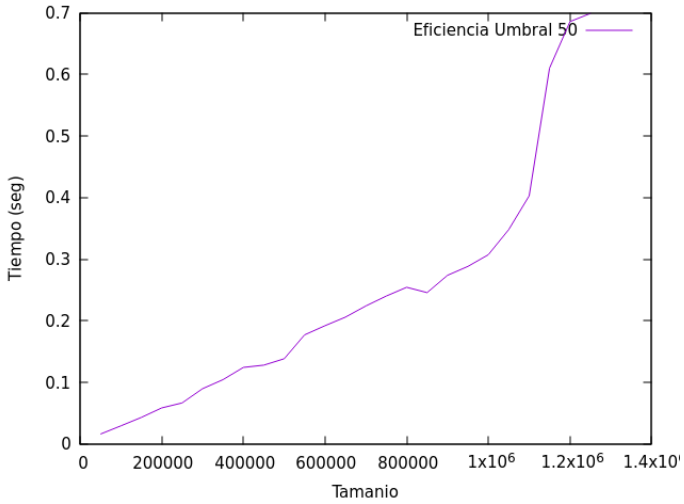

Figura 9: Grafica dyv usando UMBRAL=50

Num. puntos	Tiempo (seg)
50000	0.0184959
100000	0.0356709
150000	0.0497193
200000	0.069925
250000	0.0765743
300000	0.0981501
350000	0.114712
400000	0.135277
450000	0.154753
500000	0.153074
550000	0.181004
600000	0.201374
650000	0.220293
700000	0.244379
750000	0.25781
800000	0.387716
850000	0.420059
900000	0.444331
950000	0.46924
1000000	0.493394
1050000	0.56041
1100000	0.606963
1150000	0.658092
1200000	1.23809
1250000	1.33326

Cuadro 8: Umbral =100

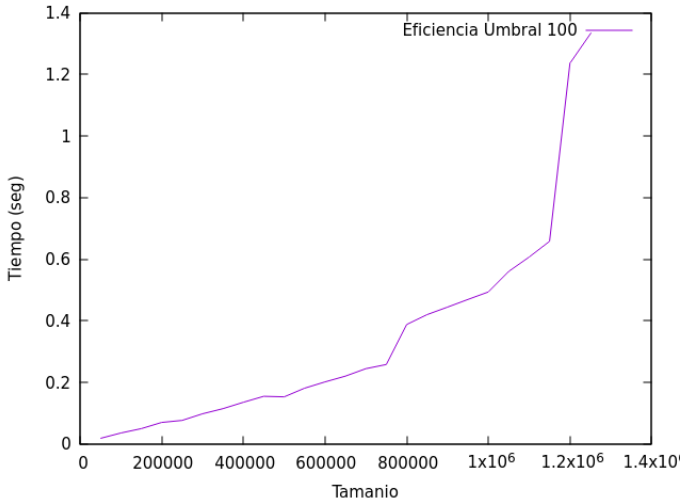

Figura 10: Grafica dyv usando UMBRAL=100

Num. puntos	Tiempo (seg)
50000	0.0614563
100000	0.115068
150000	0.139558
200000	0.224789
250000	0.346119
300000	0.260926
350000	0.358028
400000	0.432816
450000	0.586449
500000	0.951206
550000	0.696232
600000	0.808216
650000	0.888805
700000	1.08785
750000	1.13484
800000	1.3672
850000	1.52541
900000	1.63948
950000	1.64356
1000000	1.64421
1050000	1.55136
1100000	1.5743
1150000	1.62655
1200000	1.66875
1250000	1.70011

Cuadro 9: Umbral =1000

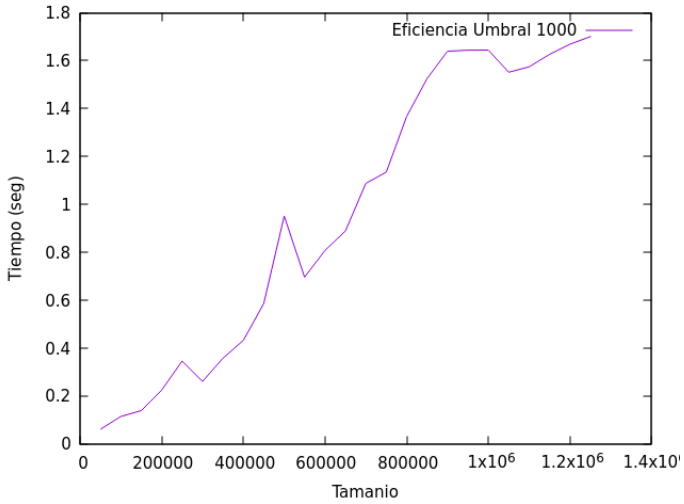

Figura 11: Grafica dyv usando UMBRAL=1000

Num. puntos	Tiempo (seg)
50000	0.247893
100000	0.457227
150000	0.984173
200000	0.831884
250000	1.22869
300000	1.71613
350000	1.15753
400000	1.45582
450000	1.78377
500000	3.14844
550000	4.19254
600000	4.76026
650000	2.81104
700000	3.18333
750000	3.85818
800000	3.96394
850000	4.41386
900000	5.02101
950000	5.55905
1000000	6.29035
1050000	6.69834
1100000	7.49271
1150000	7.73569
1200000	8.48771
1250000	9.04461

Cuadro 10: Umbral = 5000

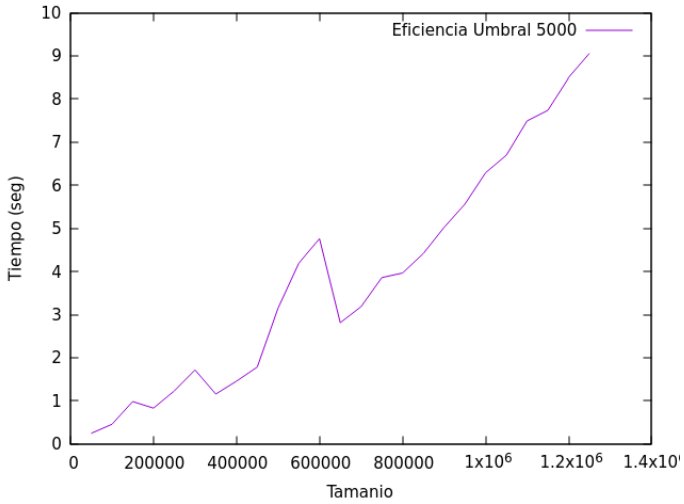

Figura 12: Grafica dyv usando UMBRAL=5000

Ahora vamos a verlas todas en una misma gráfica, vamos a poder comprobar como, cuanto mayor es el umbral, menos eficiente es nuestro programa, ya que estamos usando en mayor medida el algoritmo específico, que como ya hemos visto, es menos eficiente que el algoritmo divide y vencerás.

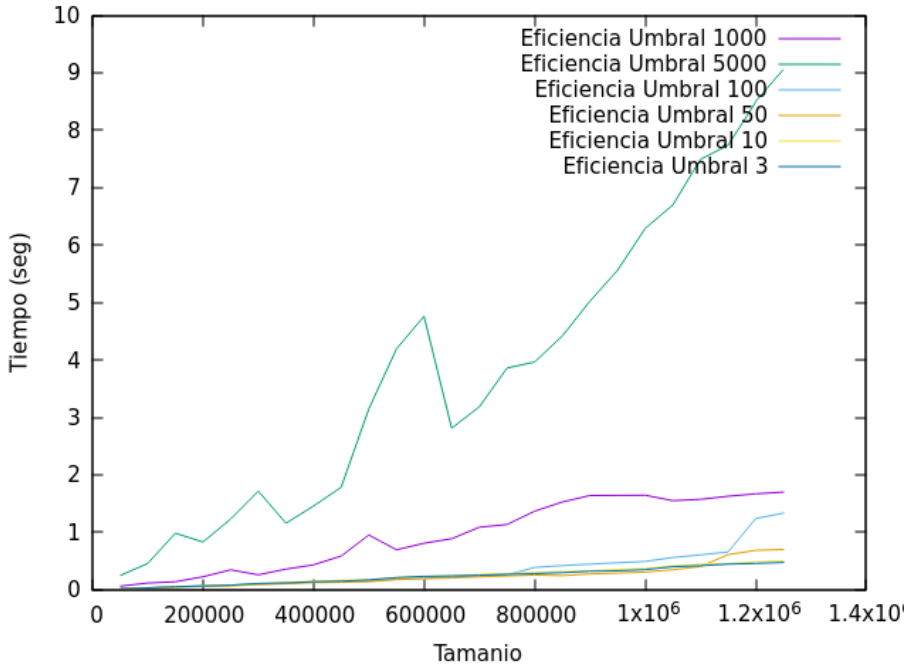

Figura 13: Grafica dyv comparación de todos los umbrales de tanteo

Luego se ha quedado más que probado que estamos antes un caso de problema donde el algoritmo divide y vencerás es siempre mejor, luego debería escogerse siempre como manera de resolver el problema en cualquier circunstancia.

7. Posiciones alcanzables

Otra de las tareas de esta práctica era mostrar para cada punto las posiciones concretas que alcanzaba. Para ello se debe de hacer primero una modificación en el struct, puse se debe de añadir una lista en la que se guarden los puntos que alcanza cada punto de manera que si un punto $p1$ alcanza a un punto $p2$ en la lista de $p1$ se guarde el punto $p2$. El struct *Punto* queda de la siguiente manera:

```
struct Punto{

    double x;
    double y;
    int pos_alcanzables;
    list<Punto> puntos;

    Punto(double i, double j){
        x=i;
        y=j;
        pos_alcanzables=0;
    }

    bool operator==(Punto P1){
        return (P1.x == this->x && P1.y == this->y);
    }

    bool operator<(const Punto& p2) const{
        bool menor = false;

        if(this->x < p2.x && this->y < p2.y){
            menor=true;
        }

        return menor;
    }
```

En el caso del algoritmo base es sencillo de implementar pues se comparan todos los puntos entre sí, en el momento en que la función *alcanza* devuelva true, además de aumentar el número de posiciones alcanzables, añadimos el punto que se alcanza a la lista correspondiente. Quedando el código de dicha función de la siguiente manera.

```
void dyv_basico (vector<Punto> &a) {

    //Recorremos todos los puntos y comprobamos si se alcanzan
    for (int x = 0; x < a.size(); x++) {
        for (int y = 0; y < a.size(); y++) {
            if (x != y && punto_alcanza(a[x], a[y])){
                a[x].pos_alcanzables++;
                a[x].puntos.insert(a[x].puntos.end(), a[y]);
            }
        }
    }
}
```

Sin embargo, no hemos sido capaces de dar con al solución correcta en el caso del algoritmo Divide y Vencerás para que además de contabilizar los puntos alcanzados, los almacene y muestre sin interferir en la eficiencia del algoritmo. Se va a mostrar una solución que resuelve el problema tal y como se pidió, pero cuya eficiencia no es la idónea pues la función de fusión tiene una eficiencia cuadrática debido a la copia de listas. Probamos distintas alternativas de estructuras de datos para almacenar estos puntos, pero ninguno realizaba todas las operaciones queridas con una eficiencia óptima. A continuación, se muestra el código que devuelve las posiciones alcanzables, pero con una eficiencia cuadrática.

```
vector<Punto> Fusion (const vector<Punto> &izq,
                        const vector<Punto> &dcha){

    // Creo un vector donde guardar la solución a devolver
    vector<Punto> solucion;

    int tam_izq = izq.size();
    int tam_dcha = dcha.size();

    //El max de las coordenadas x del vector de la izq
    double max_izq=max_element(izq.begin(),izq.end(), ordenar_x)->x;

    int i = 0; //posicion vector izquierda
    int d = 0; //posicion vector derecha
    int alcanzables = 0;
    int alcanzables_estricto= 0;


    list<Punto> ptos_alcanzados;
    list<Punto> ptos_alcanzados_estricto ;
    Punto pto_dcha = dcha.at(d);
    Punto pto_izq = izq.at(i);

    do{
        // Tomamos un punto de cada vector
        if (d < tam_dcha) {
            pto_dcha = dcha.at(d);
        }
        if (i < tam_izq) {
            pto_izq = izq.at(i);
        }

        bool en_lim = (d == tam_dcha || i == tam_izq);

        if (en_lim) {
            if (d == tam_dcha) {
                while (i < tam_izq) {
                    solucion.push_back(izq.at(i));
                    alcanzables++;
                    Punto aux = izq.at(i);
                    // Añadimos el punto alcanzado a la lista
                    ptos_alcanzados.insert(
                                ptos_alcanzados.end(),izq.at(i));
                    if (izq.at(i).x != max_izq) {
                        alcanzables_estricto++;
                        ptos_alcanzados_estricto.insert(
                            ptos_alcanzados_estricto.end()izq.at(i));
                    }
                    i++;
                }
            }
```

```
        else {
            if (i == tam_izq) {
                while (d < tam_dcha) {
                    solucion.push_back(dcha.at(d));
                    if (dcha.at(d).x == max_izq) {
                        solucion.back().pos_alcanzables +=
                                        alcanzables_estricto;
                        list<Punto> copia=ptos_alcanzados_estricto;
                        solucion.back().puntos.splice
                                solucion.back().puntos.end(),copia);
                    } else {
                        solucion.back().pos_alcanzables +=
                                            alcanzables;
                        list<Punto> copia = ptos_alcanzados;
                        solucion.back().puntos.splice
                            (solucion.back().puntos.end(),copia);
                    }
                    d++;
                }
            }
        }
    }
    else {
        if ((pto_izq.y < pto_dcha.y)) {
            solucion.push_back(pto_izq);
            alcanzables++;
            ptos_alcanzados.insert(ptos_alcanzados.end(), pto_izq);
            if (pto_izq.x != max_izq) {
                alcanzables_estricto++;
                ptos_alcanzados_estricto.insert
                        (ptos_alcanzados_estricto.end(), pto_izq);
            }
            i++;
        }
        if (pto_izq.y >= pto_dcha.y) {
            solucion.push_back(pto_dcha);
            if (pto_dcha.x == max_izq) {
                solucion.back().pos_alcanzables +=
                                alcanzables_estricto;
                list<Punto> copia = ptos_alcanzados_estricto;
                solucion.back().puntos.splice
                        (solucion.back().puntos.end(),copia);
            } else {
                solucion.back().pos_alcanzables += alcanzables;
                list<Punto> copia = ptos_alcanzados;
                solucion.back().puntos.splice
                    (solucion.back().puntos.end(),copia);
            }
            d++;
        }
    }
}while(d<tam_dcha||i<tam_izq);
return solucion;
}
```

8. Conclusiones

A continuación se van a exponer las conclusiones y resultados a los que se han llegado con el desarrollo de la práctica. La primera de ellas y quizás más evidente es como en este caso concreto; es decir, para este problema en particular, se ha probado como el uso de la técnica Divide y Vencerás es mejor para cualquier tamaño del problema.

En ocasiones, es esperable que por debajo de umbral, el algoritmo específico reduzca los tiempos de ejecución; esto es, para que tamaño de problemas pequeños, el método iterativo sea más eficiente, y que conforme aumenta el tamaño del problema, en este caso, el número de puntos, el algoritmo divide y vencerás acabe superando al recursivo, y este último se convierta en la mejor opción para problemas de tamaño superior. En estos casos, se define la resolución del problema en función del umbral; es decir, para tamaños menores que el umbral se aplica el modelo especifico y para tamaños mayores el recursivo. Esto también puede suceder porque el recursivo usa más pila y puede llevar a que sea menos eficiente.

Sin embargo, en este caso se ha comprobado mediante las tablas de tiempos, las gráficas y los cálculos del umbral que el método recursivo ofrece siempre unos mejores resultados de tiempo que el algoritmo específico. Se ve fácilmente comparando las tablas de tiempos y las gráficas como el algoritmo iterativo es siempre peor y ha sido a través del estudio de los umbrales donde hemos podido ver que independientemente de si el umbral tomado se acercaba o no al calculado como óptimo, la conclusión seguía siendo la misma.