UNIVERSIDAD DE GRANADA

E.T.S DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Algorítmica. Práctica 3: Algoritmos Greedy.

Miembros del grupo 3:
Benaisa Cruz, Hamed Ignacio 75932445S
Espínola Pérez, Sergio 78006823T
Feixas Galdeano, José Miguel, 20080844G
Ruiz de Valdivia Torres, David Jesús 78006825W

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Índice:

1. Algoritmo cercanía

1.1 Introducción

En este algoritmo se traza la ruta buscando siempre el nodo más cercano respecto al que estamos.

1.12 Eficiencia teórica

Para analizar la eficiencia teórica aplicamos la regla de la suma:

- Bloque 1

```
list<int> candidatos;
```

La eficiencia es de O(1).

- Bloque 2

```
for(int i=0; i<dimension; i++)
    candidatos.push_back(i+1);
```

La eficiencia de rellenar la lista de candidatos es de O(n).

- Bloque 3

```
result.push_back(candidatos.front());
candidatos.pop_front();

list<int>::iterator it_candidatos;
list<int>::iterator it_resultado = result.begin();
double dist_min;
list<int>::iterator menor_candidato;
```

Tanto la operación de push_back como las inicializaciones de variables como los accesos a memoria tienen una complejidad constante. Por tanto la eficiencia de este bloque es de $O(7) \in O(1)$.

- Bloque 4

```
while ( !candidatos.empty() )
{
    it_candidatos = candidatos.begin();
    dist_min = matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1];
    menor_candidato = it_candidatos;

    for(; it_candidatos != candidatos.end(); ++it_candidatos){
        if(dist_min > matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1])
        {
            dist_min = matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1];
            menor_candidato = it_candidatos;
        }
    }

    result.push_back( (*menor_candidato) );
    candidatos.erase( menor_candidato );

   ++it_resultado;
}
```

Aplicamos la regla del producto sobre el while exterior, y la regla de la suma sobre este subbloque de código que encapsula el while.

- Bloque 4.1

```
it_candidatos = candidatos.begin();
dist_min = matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1];
menor_candidato = it_candidatos;
```

La eficiencia teórica de este bloque es de O(6) ∈ O(1).

- Bloque 4.2

```
for(; it_candidatos != candidatos.end(); ++it_candidatos){
    if(dist_min > matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1])
    {
        dist_min = matrizAdyacencia[(*it_resultado)-1][(*it_candidatos)-1];
        menor_candidato = it_candidatos;
    }
}
```

Este bloque de código tiene una eficiencia de O(n), ya que en el peor caso se ejecuta el interior del if, que tiene una eficiencia de O(1), desde 0 hasta n-1 veces.

- Bloque 4.3

```
result.push_back( (*menor_candidato) );
candidatos.erase( menor_candidato );

++it_resultado;
```

Las operaciones atómicas de hacer un push_back y de erase de un solo elemento tienen una eficiencia de O(1), junto al incremento, así que este bloque tiene un O(3) ∈ O(1).

Por tanto, como el subbloque dentro del while tiene una eficiencia de O(n) y la eficiencia del propio tiene a su vez una eficiencia de O(n) aplicando la regla del producto el bloque 4 tiene una eficiencia de $O(n^2)$.

Eficiencia final

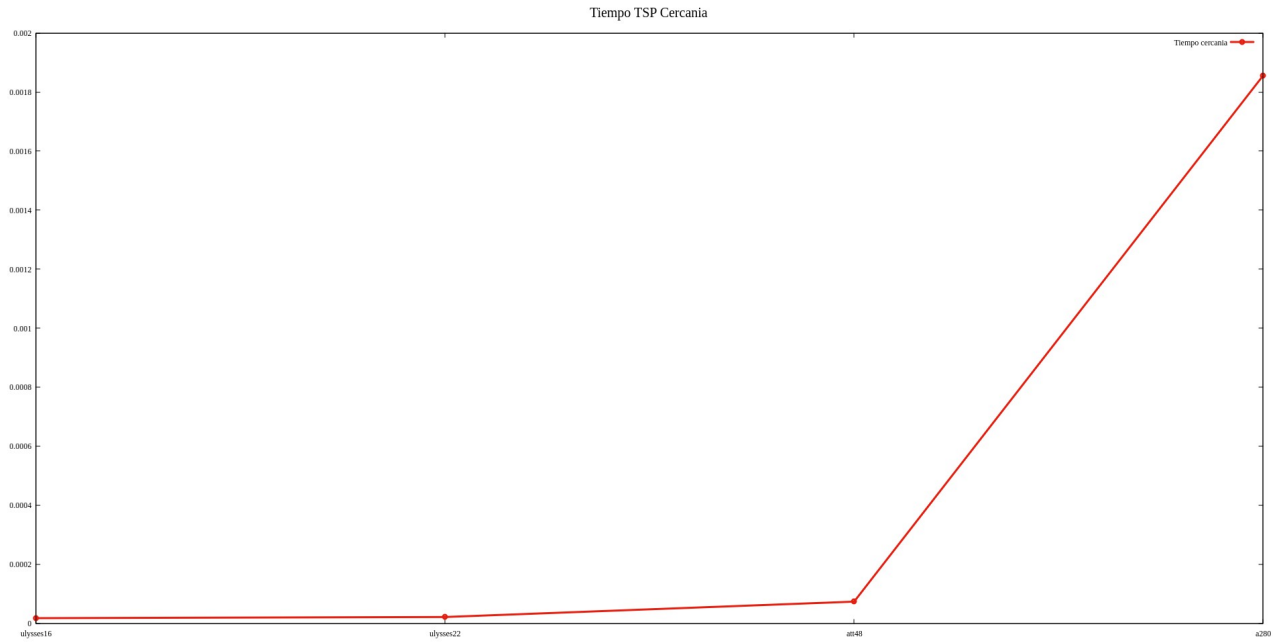
Aplicando la regla de la suma concluimos que este algoritmo tiene una eficiencia teórica de $O(n^2)$.

1.3 Eficiencia empírica

Para esta parte hemos obtenido una gráfica a partir de los tiempos de ejecución con los distintos tsp dados.
Los datos obtenidos han sido los siguientes:

TSP	Tiempo
ulysses16	1.7e-05
ulysses22	2.1e-05
att48	7.3e-05
a280	0.001856

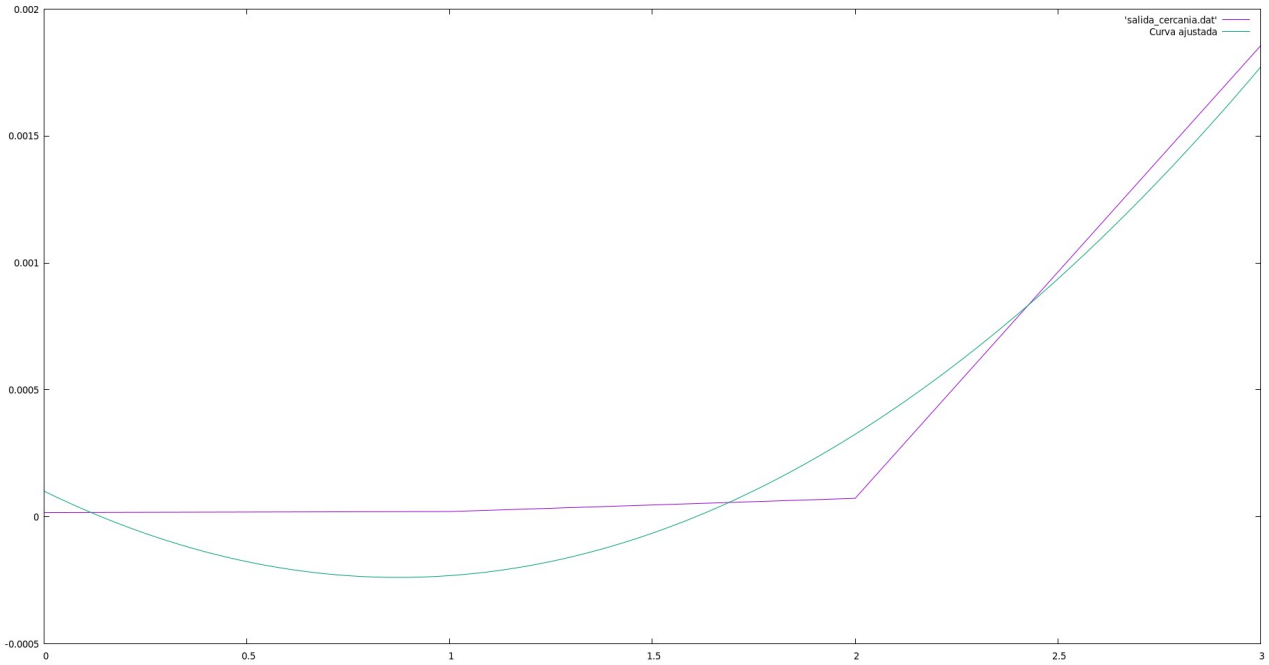
Y la gráfica resultante ha sido:



1.4 Eficiencia híbrida

Al tener una eficiencia cuadrática, el f(x) usado para sacar las constantes ocultas ha sido $a0*x^2+a1*x+a2$. Haciendo el ajuste, hemos obtenido que las constantes ocultas son:
a0 = 0.00044475
a1= -0.00077735
a2 = 0.00010115

Y tras saber esto, la curva ajustada ha sido esta:



Es un ajuste moderado.

2 Algoritmo inserción

2.1 Introducción

Este algoritmo en primera instancia forma un triángulo con los nodos más al norte, este y oeste y despúes va añadiendo los nodos que produzcan un menor incremento para trazar el camino.

2.2 Eficiencia teórica

Aplicamos la regla de la suma:

- Bloque 1

```
list<int> candidatos;
```

La declaración de una variable tiene una eficiencia de O(1).

- Bloque 2

```
for(int i=0; i<dimension; i++)
    candidatos.push_back(i+1);
```

La eficiencia de rellenar la lista de candidatos es de O(n).

- Bloque 3

```
list<int>::iterator it_candidatos = candidatos.begin();
list<int>::iterator punto_cardinal = it_candidatos;
```

La eficiencia de inicializar it_candidatos y de igualar punto_cardinal a it_candidatos es de O(4) ∈ O(1).

- Bloque 4

```
// El vértice más al norte
for(;it_candidatos != candidatos.end();++it_candidatos)
{
    if(nodos[(*punto_cardinal)-1].getY() < nodos[(*it_candidatos)-1].getY())
        punto_cardinal = it_candidatos;
}

Nodo norte = nodos[(*punto_cardinal)-1];
candidatos.erase(punto_cardinal);
punto_cardinal = it_candidatos = candidatos.begin();
```

El for inicial tiene una complejidad lineal, O(n). En cambio las tres líneas siguientes tienen una eficiencia de O(5) ∈ O(1). Por tanto este bloque de código tiene una complejidad de O(n). Este trozo de código selecciona el vértice más al norte de los vértices candidatos. Análogamente, y puesto que siguen la misma estructura, los bloques de código que seleccionan el vértice más al oeste y más al este también tienen una complejidad lineal, O(n).

- Bloque 5

```
vector<Arista> lados;
lados.push_back(Arista(norte, este));
lados.push_back(Arista(este,oeste));
lados.push_back(Arista(oeste,norte));
```

La creación del contenedor lados, junto a los tres push_back de las aristas tienen una complejidad constante.

- Bloque 6

El siguiente bloque es el while principal de la elección de nodos, es decir la parte más crítica del algoritmo de inserción. Por simplicidad vamos a aplicar a su vez la regla del producto y de la suma en sus subbloques.

- Bloque 6.1

```
int posicion_arista = 0;

list<int>::const_iterator nodo_anadir = candidatos.cbegin();
double mejor_distancia = matrizAdyacencia[lados[0].getP1().getId()-1][(*nodo_anadir)-1]
                       + matrizAdyacencia[(*nodo_anadir)-1][lados[0].getP2().getId()-1]
                       - lados[0].getDistancia();
```

La suma total de las inicializaciones de las variables tiene una eficiencia de $O(20) \in O(1)$.

- Bloque 6.2

```
for (list<int>::const_iterator cit = candidatos.cbegin(); cit != candidatos.cend(); ++cit)
{
    Nodo opcion = nodos[(*cit)-1];

    for(size_t i=0; i<lados.size(); ++i)
    {
        double incremento = matrizAdyacencia[lados[i].getP1().getId()-1][opcion.getId()-1]
                          + matrizAdyacencia[lados[i].getP2().getId()-1][opcion.getId()-1]
                          - lados[i].getDistancia();

        if (incremento < mejor_distancia)
        {
            mejor_distancia = incremento;
            nodo_anadir = cit;
            posicion_arista = i;
        }
    }
}
```

Utilizando la regla del producto podemos concluir con que la eficiencia teórica de este bloque de código es de $O(n^2)$.

- Bloque 6.3

```
Arista x(lados[posicion_arista].getP1(),nodos[(*nodo_anadir)-1]);
Arista y(nodos[(*nodo_anadir)-1],lados[posicion_arista].getP2());

vector<Arista>::const_iterator it = lados.begin() + posicion_arista +1;
lados[posicion_arista] = x;
lados.insert(it, y);

candidatos.erase(nodo_anadir);
```

Todas estas operaciones de inicialización del objeto Arista, inicialización del const_iterator it, la modificación de el elemento lados[posicion_arista], el insert de y el erase en candidatos, siguen cada una, una eficiencia teórica de O(1) por regla de la suma podemos concluir que este bloque tiene una eficiencia de O(1).

Por la regla de la suma del bloque 6 podemos decir que lo que hay dentro del while tiene una eficiencia teórica de $O(n^2)$. Por la regla del producto, el bloque 6 tiene una eficiencia teórica de $O(n^3)$.

- Bloque 7

```
for (size_t i=0; i<lados.size(); ++i)
{
    result.push_back(lados[i].getP1().getId());
}
```

La complejidad de este bloque es lineal, O(n).

Eficiencia final

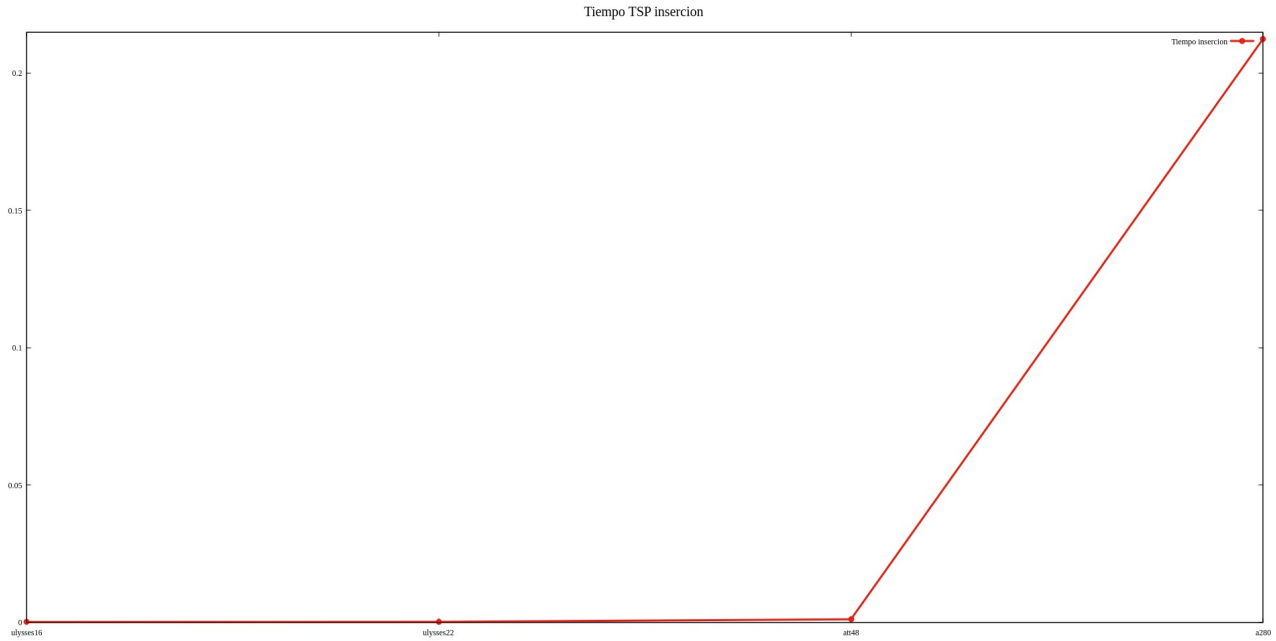
Aplicando la regla de la suma y puesto que el bloque 6 es el mayor, asindóticamente hablando, podemos concluir con que este algoritmo tiene una eficiencia teórica de $O(n^3)$.

2.3 Eficiencia empírica

Los datos obtenidos en las mediciones han sido:

TSP	Tiempo
ulysses16	6.3e-05
ulysses22	0.00013
att48	0.001061
a280	0.21256

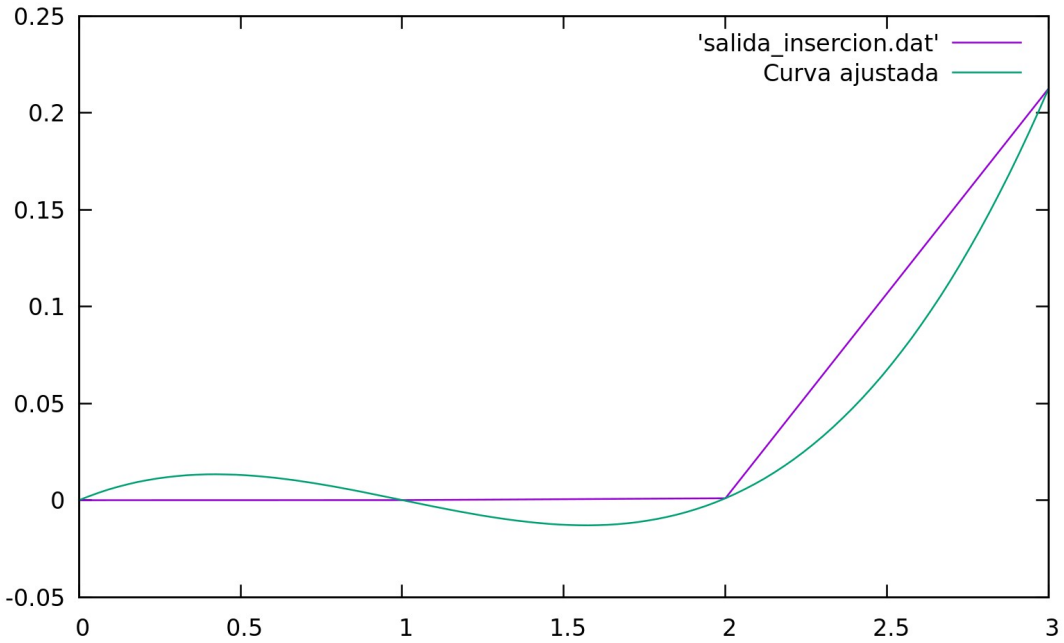
Con esto, la gráfica resultante es:



2.4 Eficiencia híbrida

Al ser de orden cúbico, el f(x) con el que se realizará el ajuste será $a0*x^3+a1*x^2+a2*x+a3$. Las constantes ocultas entonces son:
a0 = 0.0349507
a1= -0.10442
a2 = 0.0695363
a3= 6.3e-05

Y la curva ajustada con esto es la siguiente:



Es un ajuste moderadamente bueno.

3 Algoritmo espiral

3.1 Introducción

Este algoritmo parte del primer nodo de la lista. Y empieza a buscar el nodo que esté más al noroeste y los va añadiendo, si no hay más, empiezas a añadir los nodos que estén más al sureste, hasta que no quedan dónde vuelves a buscar nodos al noroeste, esto se repite hasta agotar los candidatos. Esto resulta en un recorrido que recuerda a una espiral.

3.2 Eficiencia teórica

Para analizar la eficiencia aplicamos la regla de la suma, que dice que la eficencia total es la máxima eficiencia de cada uno de los bloques.

- Bloque 1

```
list<int> izquierda;
list<int> derecha;
```

La eficiencia de la creación de dos listas de enteras vacias es O(2)∈ O(1).

- Bloque 2

```
for(int i=0; i<dimension; i++)
{
    izquierda.push_back(i+1);
}
```

Para calcular la eficiencia debemos decir que a la dimensión la vamos a llamar n, y que para ello vamos a calcular la regla del producto que dice que la eficiencia es el producto de las eficiencias.
La primera eficiencia es la de las iteraciones que obtenemos que se hacen de 0 a n-1 iteraciones, que esta sumatoria produce como resultado O(n).
Realizar un push_back en una lista sigue una eficiencia de O(1).
La eficiencia total es de O(n*1) = O(n)

- Bloque 3

```
result.push_back(izquierda.front());

list<int>::iterator it_izq = izquierda.begin();

int elementoY = izquierda.front();

it_izq = izquierda.begin();
izquierda.erase(it_izq);
it_izq = izquierda.begin();
list<int>::iterator it_elemY;

int elemento_izquierda;
int elemento_derecha;
```

Todas estas operaciones son accesos de memoria (5 oper), igualación de datos (4)y reservas de memoria (7 oper). Además se realiza un push_back que tiene O(1)y un erase de una lista, que este tiene O(1)
Por lo que la eficiencia total es de O(5+4+7+1+1) ∈ O(1).

- Bloque 4

```
while (!izquierda.empty() || !derecha.empty())
    {

        while ( !izquierda.empty())
        {
            elemento_izquierda = elementoY;

            for(;it_izq != izquierda.end();)
            {
                if(nodos[elemento_izquierda-1].getX() < nodos[(*it_izq)-1].getX())
                {
                    derecha.push_back((*it_izq));
                    it_izq = izquierda.erase(it_izq);
                }
                else
                    ++it_izq;
            }

            it_elemY = izquierda.begin();
            elementoY = izquierda.front();

            for( it_izq = izquierda.begin(); it_izq != izquierda.end();++it_izq){
                if(nodos[elementoY-1].getY() < nodos[(*it_izq)-1].getY())
                {
                    elementoY = (*it_izq);
                    it_elemY = it_izq;
                }
            }
            if(izquierda.size() != 0) {
                result.push_back((*it_elemY));
                izquierda.erase(it_elemY);
            }

            it_izq = izquierda.begin();
        }
```

```
        list<int>::iterator it_derecha = derecha.begin();

        while ( !derecha.empty())
        {
            elemento_derecha = elementoY;

            for(;it_derecha != derecha.end();)
            {
                if(nodos[elemento_derecha-1].getX() >= nodos[(*it_derecha)-1].getX())
                {
                    izquierda.push_back((*it_derecha));
                    it_derecha = derecha.erase(it_derecha);
                }
                else
                    ++it_derecha;
            }

            it_elemY = derecha.begin();
            elementoY = derecha.front();

            for( it_derecha = derecha.begin(); it_derecha != derecha.end();++it_derecha){
                if(nodos[elementoY-1].getY() > nodos[(*it_derecha)-1].getY())
                {
                    elementoY = (*it_derecha);
                    it_elemY = it_derecha;
                }
            }
            if (derecha.size() != 0) {
                result.push_back((*it_elemY));
                derecha.erase(it_elemY);
            }
            it_derecha = derecha.begin();
        }
    }
```

Para analizar este bloque debemos realizar una regla de la suma sobre cada sub-bloque, y luego una regla del producto con el bucle total.

- Bloque 4.1

```
while ( !izquierda.empty())
        {
            elemento_izquierda = elementoY;

            for(;it_izq != izquierda.end();)
            {
                if(nodos[elemento_izquierda-1].getX() < nodos[(*it_izq)-1].getX())
                {
                    derecha.push_back((*it_izq));
                    it_izq = izquierda.erase(it_izq);
                }
                else
                    ++it_izq;
            }

            it_elemY = izquierda.begin();
            elementoY = izquierda.front();
```

```
            for( it_izq = izquierda.begin(); it_izq != izquierda.end();++it_izq){
                if(nodos[elementoY-1].getY() < nodos[(*it_izq)-1].getY())
                {
                    elementoY = (*it_izq);
                    it_elemY = it_izq;
                }
            }
            if(izquierda.size() != 0) {
                result.push_back((*it_elemY));
                izquierda.erase(it_elemY);
            }

            it_izq = izquierda.begin();
        }
```

Con este bloque también se debe realizar una regla de la suma primero, para luego realizar la regla del producto.

- Bloque 4.1.1

```
elemento_izquierda = elementoY;
```

En este bloque se hace un acceso de memoria (1 oper) y se igualan datos una vez (1 oper), es decir, que tiene O(2)
$\in$ O(1).

- Bloque 4.1.2

```
for(;it_izq != izquierda.end();)
{
    if(nodos[elemento_izquierda-1].getX() < nodos[(*it_izq)-1].getX())
    {
        derecha.push_back((*it_izq));
        it_izq = izquierda.erase(it_izq);
    }
    else
        ++it_izq;
}
```

it_izq siempre empieza en el begin de izquierda, y en el peor caso izquierda tendrá n elementos, es decir que recorrerá la lista desde la posición 0 a la n-1.
El if siempre debe comprobar la condición que está formad por cuatro accesos a memoria (4 oper), dos restas (2 oper) y una comprobación de si es menor (1 oper).
Dentro del if se hace una asignación(1 oper), tres accesos a memoria (3 oper), un push_back (O(1)) y un erase (O(1)).
La peor eficiencia dentro del bucle pertenece a O(1), como se hacen iteraciones de 0 a n en el peor caso, la función que seguiria esta eficiencia

sería una sumatoria desde 0 hasta n-1, de unos, lo que da n, es decir, que la eficiencia es O(n).

- Bloque 4.1.3

```
it_elemY = izquierda.begin();
elementoY = izquierda.front();
```

- Bloque 4.1.4

```
for( it_izq = izquierda.begin(); it_izq != izquierda.end();++it_izq){
    if(nodos[elementoY-1].getY() < nodos[(*it_izq)-1].getY())
    {
        elementoY = (*it_izq);
        it_elemY = it_izq;
    }
}
```

Aplicamos la regla del producto.
El bucle va desde el elemento 1 al n, las operaciones que se hacen en el bucle por si solo son una suma (1 oper), un operador lógico (1 oper) y una asignación (1 oper), es decir que se hacen 3n operaciones por lo que la eficiencia es O(3n) $\in$ O(n)
El interior del cuerpo, en el peor caso debe comprobar la condición formada por: cuatro accesos de memoria (4 oper), dos restas (2 oper), y un operador de orden (1 oper). Dentro del cuerpo del if se hacen tres accesos a memoria y dos asignaciones, por lo que la eficiencia es O(12) $\in$ O(1).

Aplicando la regla del producto obtenemos que O(n*1) = O(n)

- Bloque 4.1.5

```
if(izquierda.size() != 0) {
    result.push_back((*it_elemY));
    izquierda.erase(it_elemY);
}
```

En este if se hacen cuatro accesos a memoria (4 oper), una operación lógica (1 oper), se llama una vez a la operación push_back (O(1)) y se llama a la función erase (O(1)).
La eficiencia total es del orden de constante O(1).

- Bloque 4.1.6

```
it_izq = izquierda.begin();
```

Se hace una asignación y dos accesos de memoria (2 oper), por lo que pertenece al orden de O(1).

Fin del bloque 4.1
Al aplicar la regla de la suma obtenemos que la mayor eficiencia es O(n).
En el peor caso todos los elementos siempre van a estar a la izquierda, por lo que ira eliminando elemento a elemento,es decir, ira desde n-1 elementos a 0, por lo que la eficiencia de este bloque será O(n*n) = O(n2)

Bloque 4.2
Este bloque tiene el mismo código salvo porque utiliza las variables con el nombre derecha, y las condiciones son las contraria, pero la eficiencia es la misma, por lo que será O(n2).

Fin bloque 4
Según la regla de la suma, el máximo de eficiencia es O(n2).
En el peor caso para el bucle interior se dividirá la cantidad de elementos en dos, por lo que los incrementos se harán de dos en dos, ya que se borraran como mínimo dos elementos por iteración, lo que nos daría una eficiencia de O(log n). Pero el peor caso que supisimos anteriormente es que todos los elementos están en la izquierda o la derecha lo que haría que el bucle mayor solo haga una iteración resultando en una eficiencia de O(n2)

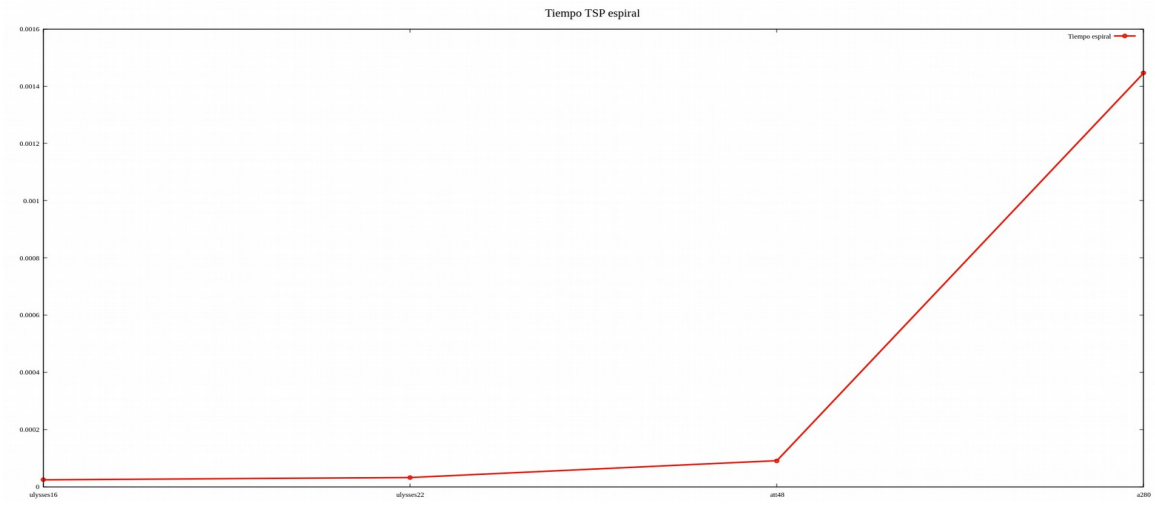
Aplicando la regla del producto obtenemos O(n2)

Eficiencia final
La eficiencia máxima obtenida en cada bloque es O(n2).

3.3 Eficiencia empírica
Estos son los datos obtenidos:

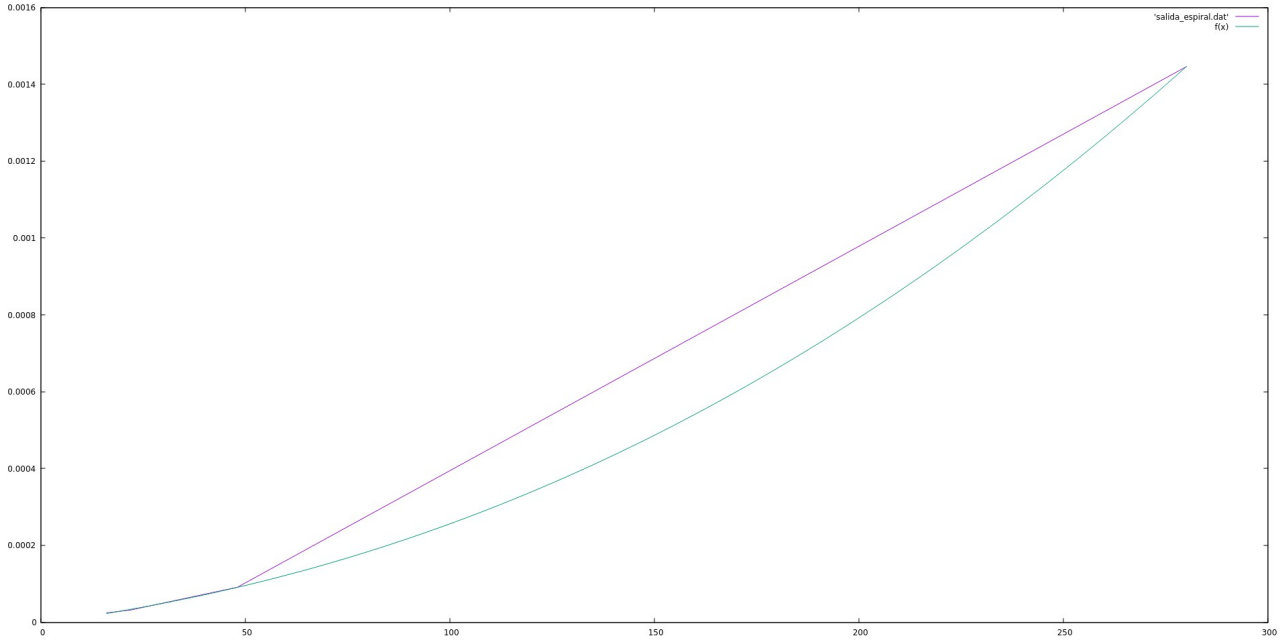
TSP	Tiempo
ulysses16	2.4e-05
ulysses22	3.2e-05
att48	9.1e-05
a280	0.001446

Y esta la gráfica obtenida:



3.4 Eficiencia híbrida

f(x) aquí sería a0*x*x+a1*x+a2 y las constantes obtenidas son:
a0 = 2.31e-09
a1 = 1.53219e-06
a2 = -3.49016e-06
Y la curva ajustada quedaría así:



Es un buen ajuste.

4 Algoritmo buques (maxContenedores)

4.1 Introducción

Este algoritmo maximiza el número de contenedores eligiendo siempre el menos pesado.

4.2 Eficiencia teórica

Para analizar el código aplicamos la regla de la suma, es decir, la eficiencia total sera la máxima eficiencia de cada bloque de código.

- Bloque 1:

```
vector<pair<int,double>> result;
```

La creación de un vector de pair equivale a la asignación de memoria de dos datos, es decir, dos operaciones por lo que O(2) ∈ O(1).

- Bloque 2:

```
vector<pair<int,double>> candidatos = pesos;
```

Si decimos que pesos tiene n cantidad de pair se deben hacer 2n reservas de memoria para candidatos, y luego 2n datos que se igualan, esto supone O(2n+2n) = O(4n) ∈ O(n).

- Bloque 3:

```
double carga_actual = 0;
```

Reservar memoria para un double (1 operación) y luego asignarlo a 0 (1 operacion), por lo que O(2) ∈ O(1).

- Bloque 4

```
vector<pair<int,double>>::iterator it_candidatos = candidatos.begin();
```

Un iterador al final es un puntero, por lo que realmente es igualar un puntero a otro (2 operaciones), por lo que O(2) ∈ O(1).

- Bloque 5

```
while ( !candidatos.empty() )
    {
        vector<pair<int,double>>::iterator it = candidatos.begin();
        for (unsigned int i=0; i<candidatos.size() && it != candidatos.end(); ++i, ++it)
        {
            if ( (*it).second < (*it_candidatos).second)
                it_candidatos = it;
        }

        if((carga_actual + (*it_candidatos).second) <= k){
            carga_actual += (*it_candidatos).second;
            result.push_back(*it_candidatos);
            candidatos.erase(it_candidatos);
            it_candidatos = candidatos.begin();
        } else
            candidatos.erase(it_candidatos);
    }
```

Para encontrar la eficiencia de este bloque de código aplicamos la regla del producto, que nos dice que su eficiencia es el producto de las eficiencia del bloque que lo compone (para ello aplicamos la regla de la suma) con la del bloque iterativo.

Como en cada iteración como máximo se reduce en uno el número de candidatos podemos asegurar que es una sumatoria desde 0 a n de las eficiencia de dentro.

- Bloque 5.1

```
vector<pair<int,double>>::iterator it = candidatos.begin();
```

Como dijimos un iterador es un puntero, por lo que realmente es igualar un puntero a otro (2 operaciones, por lo que $O(2) \in O(1)$).

- Bloque 5.2

```
for (unsigned int i=0; i<candidatos.size() && it != candidatos.end(); ++i, ++it)
        {
            if ( (*it).second < (*it_candidatos).second)
                it_candidatos = it;
        }
```

Aplicamos la regla del producto.
El for, en el peor caso, tendra que recorrer los n elementos de candidatos por lo que sera una sumatoria desde 0 a n-1 de la eficiencia del bloque del if
En el peor caso, el if se ejecuta. Dentro de este tenemos una asignacion(1 oper), la comprobacion se basa en acceder al valor de it (1 oper) y dentro de este valor al de second (1 oper), a su vez realiza lo mismo con it_candidatos (2 oper). Esto hace que la eficiencia sea de $O(5) \in O(1)$.

Como es una sumatoria de n elementos, es una sumatoria de n unos, lo que resulta en O(n).

$$\sum_{i=0}^{n-1} 1 = n$$

- Bloque 5.3

```
if((carga_actual + (*it_candidatos).second) <= k){
            carga_actual += (*it_candidatos).second;
            result.push_back(*it_candidatos);
            candidatos.erase(it_candidatos);
            it_candidatos = candidatos.begin();
        } else
            candidatos.erase(it_candidatos);
}
```

La condición siempre se debe evaluar, por lo que se hace una suma (1 oper) se accede al valor it_candidatos (1 oper) y al valor second de este (1 oper), y también se accede al valor carga_actual (1 oper) además se comprueba si es menor o igual (2 oper) y se accede al valor k (1 oper). Por lo que la condición es O(7) ∈ O(1).

En el peor caso se realiza lo que hay en el if, que para esto realizamos la regla de la suma:
El primer bloque tiene eficiencia O(5)∈ O(1)
El segundo bloque tiene complejidad O(1)
El borrado en vector tiene complejidad O(n)
El igualar dos iteradores tiene complejidad O(1)
Es decir, la eficiencia de este bloque es O(n)

Fin bloque 5
Como hemos visto la mayor eficiencia es O(n), por lo que se hacen n iteraciones de O(n), esto resulta en O(n*n iteraciones) = O(n2).

- Bloque 6

```
return result;
```

El vector result se puede haber llenado como máximo con n elementos, por lo que devolver los n elementos será O(n).

Eficiencia final

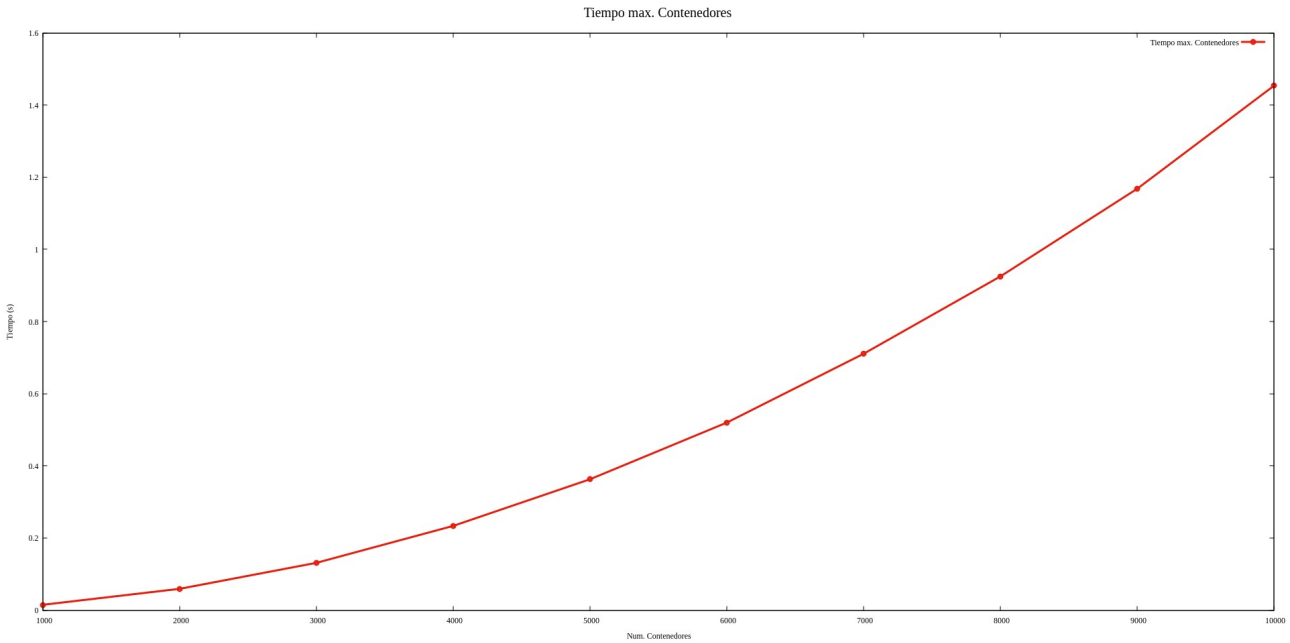
Al aplicar la regla de la suma nos sale que la eficiencia teórica máxima que se puede sacar es O(n2).

4.3 Eficiencia empírica

Los datos obtenidos son:

Tamaño	Tiempo
1000	0.014872
2000	0.059084
3000	0.131208
4000	0.233607
5000	0.362908
6000	0.520096
7000	0.710954
8000	0.925376
9000	1.16844
10000	1.45406

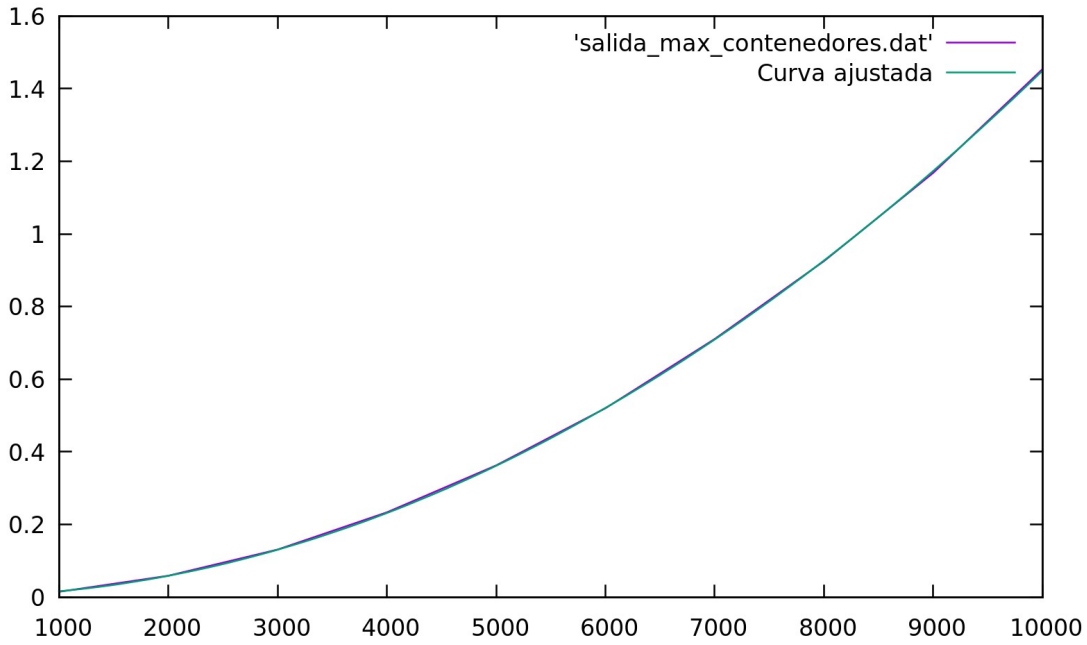
Y la correspondiente gráfica:



4.4 Eficiencia híbrida

f(x) = a0*x*x+a1*x+a2 y las constantes ocultas serán:
a0 = 1.45695e-08
a1 = -1.00274e-06
a2 = 0.00264848

Y la curva ajustada:



Es un muy buen ajuste.

5 Algoritmo buques (maxToneladas)

5.1 Introducción

Este algoritmo maximiza el número de toneladas eligiendo siempre el contenedor más pesado de los candidatos.

5.2 Eficiencia teórica

Para analizar el código aplicamos la regla de la suma, es decir, la eficiencia total será la máxima eficiencia de cada bloque de código.

- Bloque 1:

```
vector<pair<int,double>> result;
```

La creación de un vector de pair equivale a la asignación de memoria de dos datos, es decir, dos operaciones por lo que $O(2) \in O(1)$.

- Bloque 2:

```
vector<pair<int,double>> candidatos (pesos.size());
```

Si decimos que pesos tiene n cantidad de pair se deben hacer 2n reservas de memoria para candidatos, $O(2n) \in O(n)$.

- Bloque 3

```
for (unsigned int i = 0; i < candidatos.size(); i++)
    {
        candidatos[i].first = pesos[i].first;
        candidatos[i].second = pesos[i].second;
    }
```

Se accede candidatos[i] dos veces (2 oper), y luego se accede al miembro first (1 oper) y al miembro second (1 oper). Con pesos se realiza lo mismo (4 oper) y luego se realizan dos asignaciones (2 oper). Las iteraciones se hacen de 0 a n-1, es decir, que sigue una sumatoria de 8 operaciones de n a n-1. Como $O(8) \in O(1)$, es una sumatoria de 0 a n-1 de 1, lo que resulta en n.
La eficiencia de este bloque es O(n).

- Bloque 4

```
double carga_actual = 0;
double max = 0;
int indice;
int indice_max;
vector<pair<int,double>>::iterator it_candidatos;
vector<pair<int,double>>::const_iterator cit;
```

Se hacen seis declaraciones (6 oper) y dos de estas se igualan (2oper), por lo que es $O(8) \in O(1)$.

- Bloque 5

```
while ( !candidatos.empty() )
    {

        for(cit = candidatos.cbegin(), indice=0; cit != candidatos.cend(); ++cit, ++indice)
        {
            if( (*cit).second > max ){
                max = (*cit).second;
                indice_max = indice;
            }

        }

        if( (carga_actual+max) <= k){
            carga_actual += max;
            it_candidatos = (candidatos.begin()+indice_max);
            result.push_back(candidatos[indice_max]);
            candidatos.erase(it_candidatos);
```

```
        } else
        {
            it_candidatos = (candidatos.begin()+indice_max);
            candidatos.erase(it_candidatos);
        }

        max=0;


    }
```

Para saber la eficiencia aplicaremos la regla del producto, que es multiplicar la eficiencia del bucle por la eficiencia del cuerpo de este.

Para saber la eficiencia del cuerpo aplicamos regla de la suma.

- Bloque 5.1

```
for(cit = candidatos.cbegin(), indice=0; cit != candidatos.cend(); ++cit, ++indice)
        {
            if( (*cit).second > max ){
                max = (*cit).second;
                indice_max = indice;
            }

        }
```

Aplicamos la regla del producto.

El for, en el peor caso, tendrá que recorrer los n elementos de candidatos por lo que será una sumatoria desde 0 a n-1 de la eficiencia del bloque del if
En el peor caso, el if se ejecuta. Dentro de este tenemos dos asignaciones (2 oper) y tres accesos a memoria (3 oper). La comprobación se basa en acceder al valor de cit (1 oper) y dentro de este valor al de second (1 oper), y acceder al valor de max (1oper). Esto hace que la eficiencia sea de $O(8) \in O(1)$.

Como es una sumatoria de 0 a n-1 elementos, es una sumatoria de n unos, lo que resulta en O(n).

- Bloque 5.2

```
if( (carga_actual+max) <= k){
    carga_actual += max;
    it_candidatos = (candidatos.begin()+indice_max);
    result.push_back(candidatos[indice_max]);
    candidatos.erase(it_candidatos);
} else
{
    it_candidatos = (candidatos.begin()+indice_max);
    candidatos.erase(it_candidatos);
}
```

La condición siempre se comprueba esta formada por tres accesos a memoria (3 oper) y dos operaciones para comprobar si es menor o igual (2 oper).
En el peor caso entra al if, en este hay 5 accesos a memoria, 2 sumas, 1 igualación.
Se llama al método erase que sigue una eficiencia O(n).
También se llama al método push_back que este tiene O(1)
La eficiencia final, por regla de la suma es O(n).

- Bloque 5.3

```
max=0;
```

Se realiza una asignación por lo que es O(1)

Fin bloque 5
Aplicando la regla de la suma nos queda que es O(n).
Como máximo por iteración se elimina 1 elemento, por lo que el while va a ir desde n-1 a 0, por lo que esto seria O(n). Aplicando la regla del producto obtenemos que O(n*n) = O(n2).

- Bloque 6

```
return result;
```

Devolver un vector de n elementos tiene como eficiencia O(n).

Eficiencia final

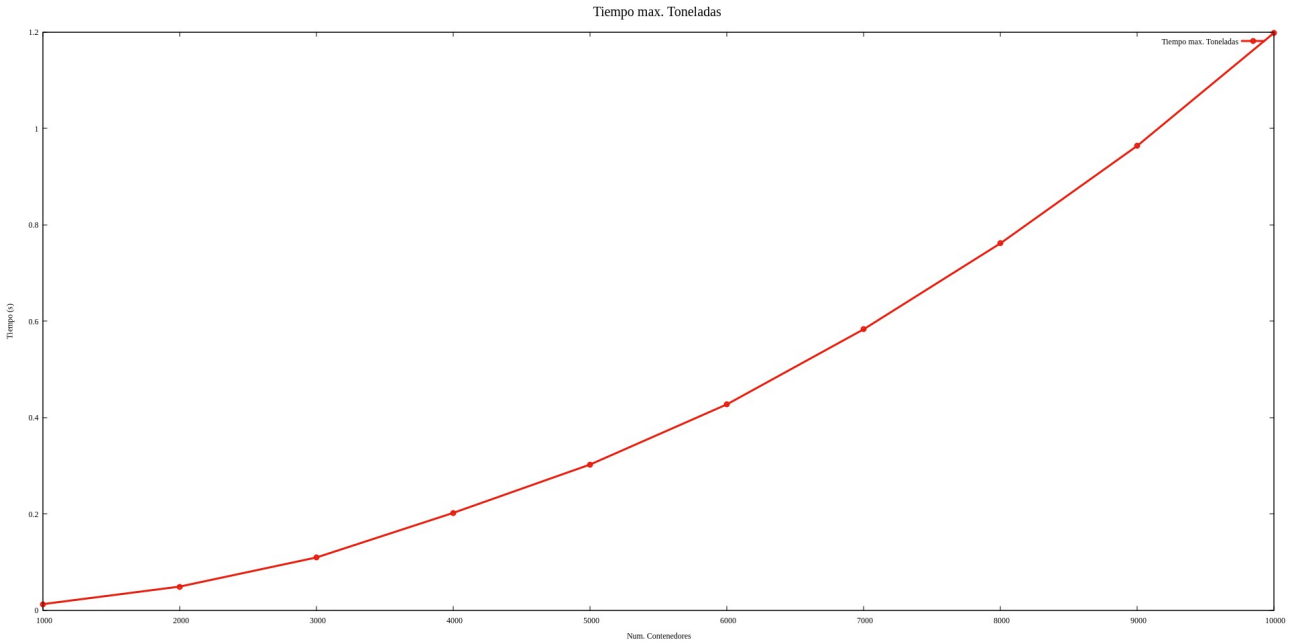
Aplicando la regla de la suma, la mayor eficiencia que tenemos es $O(n^2)$.

5.3 Eficiencia empírica

Los datos obtenidos han sido:

Tamaño	Tiempo
1000	0.012347
2000	0.048751
3000	0.109231
4000	0.201837
5000	0.302231
6000	0.427137
7000	0.583042
8000	0.761539
9000	0.963983
10000	1.19844

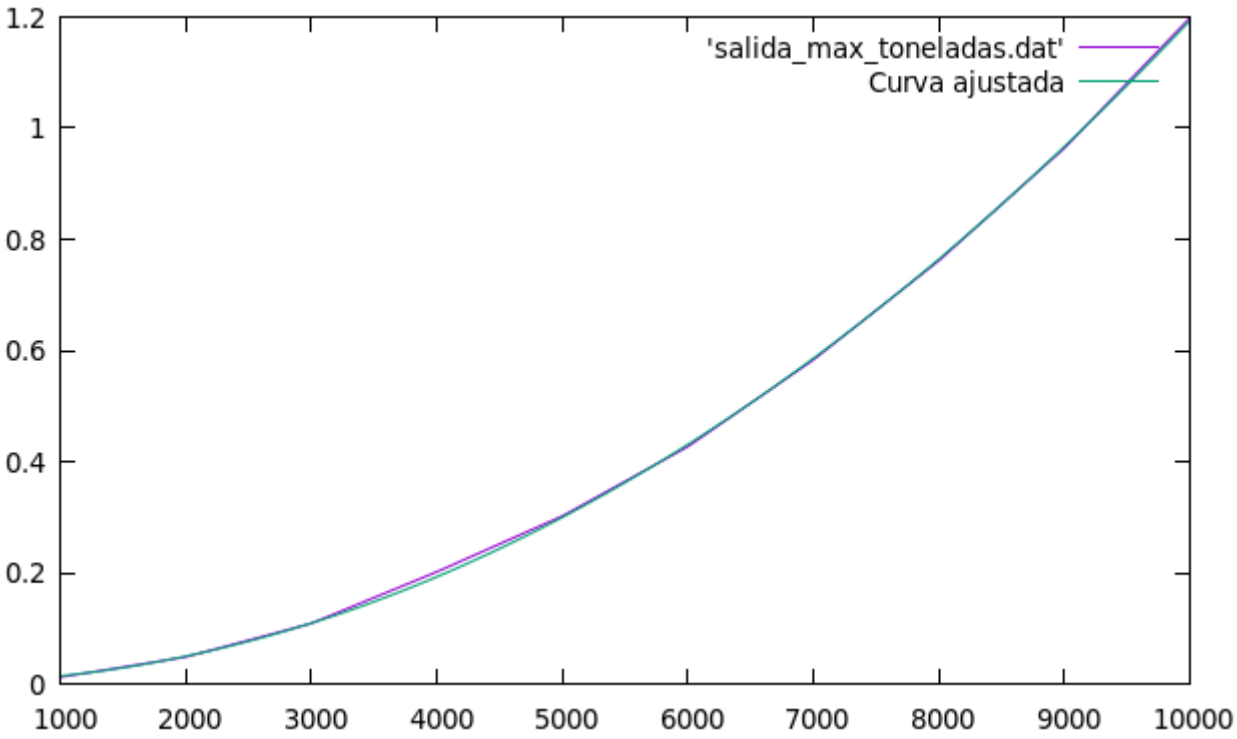
Y su gráfica correspondiente es:



5.4 Eficiencia híbrida

f(x) = a0*x*x+a1*x+a2 y sus constantes ocultas son:
a0 = 1.19216e-08
a1 = -1.59037e-07
a2 = 0.00274565

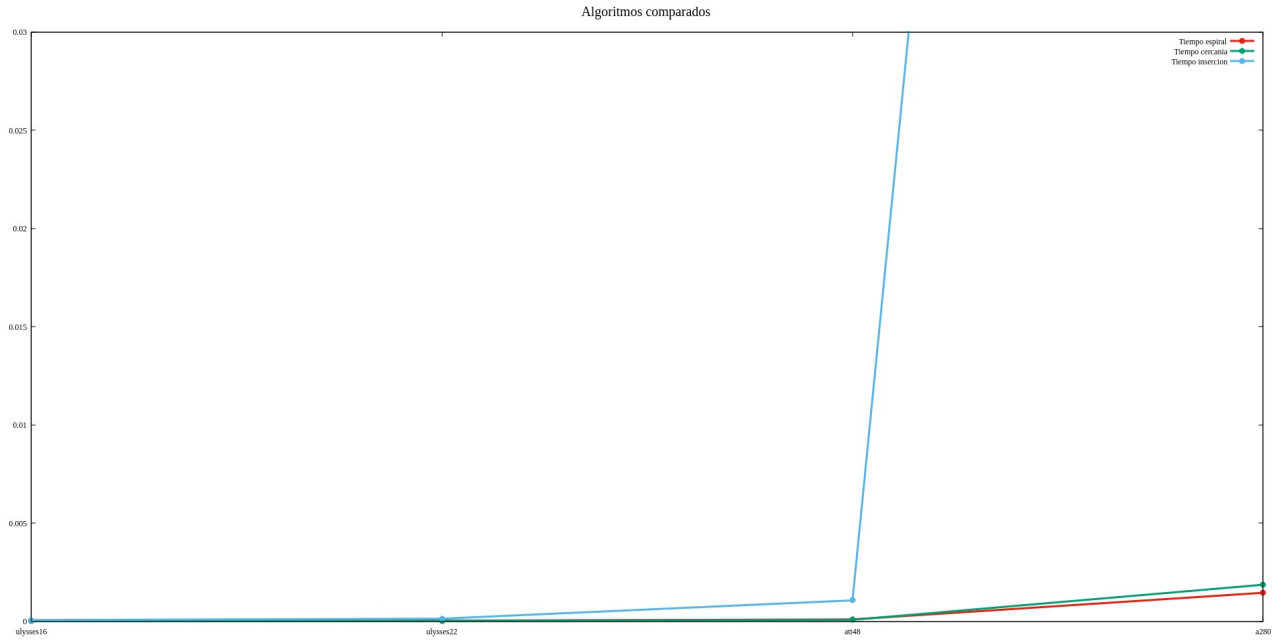
Y la curva ajustada sería:



Es un muy buen ajuste.

6 Comparativa de tiempos

Con los datos obtenidos hemos trazado la siguiente gráfica:



Podemos observar que inserción arroja notablemente peores resultados que los demás, esto es por ser cúbico, entre los dos cuadráticos podemos ver que espiral es mejor que cercanía.