UNIVERSIDAD
DE GRANADA

GRADO EN INGENIERÍA INFORMÁTICA

Práctica 3: Algoritmos Voraces (Greedy)

Autores:
Yunkai Lin Pan: 20 %
Alfonso Jesús Piñera Herrera: 20 %
Álvaro Hernández Coronel: 20 %
Jaime Castillo Uclés: 20 %
Yeray López Ramírez: 20 %

Curso:2ºC
Asignatura: Algorítmica
Fecha: 22 de Marzo de 2022
Grupo de prácticas: C2
Número de grupo: 3

Yunkai Lin Pan
Alfonso Jesús Piñera Herrera
Álvaro Hernández Coronel
Jaime Castillo Uclés
Yeray López Ramírez

Índice

1. Primera Parte: Problema de las Salas de Conferencias

1.1. Enunciado Formal del Problema

Un centro educativo va a realizar un ciclo de n conferencias durante un día. Cada conferencia tiene establecido su horario, la conferencia Si empieza a la hora Si y termina a la hora Fi. Se desea que esta actividad interfiera lo mínimo posible con las actividades normales del centro. Por tanto, se quiere diseñar un algoritmo voraz que permita planificar todas las conferencias en su

horario establecido usando el menor número de aulas posible (obviamente, dos conferencias no se pueden planificar en la misma aula si sus horarios se solapan). Demostrad la optimalidad del algoritmo.

Podemos entender del enunciado, que toda conferencia tendrá un horario fijo.

1.2. Algoritmo en pseudocódigo

A continuación, realizamos una posible implementación en pseudocódigo del algoritmo voraz que resuelve el problema.

```
1. SeleccionAulas( Conferencias C)
2. {      vector<vector<aulas>> aulas_necesarias;
3.      bool asignado = false;
4.
5.      for(int a=0; a<C.size(); a++){
6.           AsignarAConferenciaPosicion(C[a]);
   //Ordenamos el vector por horas, las conferencias que
   empiezan primero están primero
7.      }
8.
9.      for(int i=0; i<C.size(); i++){
10.          asignado = false;
11.          conferencia = C.front();
        // Sacar y eliminar un elemento del vector de
   Conferencias
12.          horario = conferencia.getHorario();
        //Guardar el horario del elemento
13.          for(int j=0; j<aulas_necesarias.size(); j++){
14.               if(NoExisteConferencia(j, horario)){
        //Comprueba que no existe una conferencia ya asignada a
   esa aula en ese horario.
15.
   aulas_necesarias[j].push_back(conferencia);     // Asigna la
   conferencia al aula.
16.                    asignado = true;
17.                    break;
18.               }
19.          }
20.          if(!asignado){
21.               Aula nueva_aula;
22.               aulas_necesarias.push_back(nueva_aula);
                    //Crea una nueva aula
```

```
23.
   aulas_necesarias[aulas_necesarias.size()-1].push_back(confere
   ncia); // Asigna la conferencia a esta nueva aula
24.            }
25.
26.      }
27.
28.      return aulas_necesarias.size();
29.   }
30.
```

Nuestro algoritmo, siguiendo su criterio de selección primeramente ordena por las conferencias que tengan menor hora de inicio, en caso de que existan dos o más conferencias que tengan iguales horas de inicio Si, se ordenarán primero las que hora de fin Fi es mayor (por tener algún criterio adicional, para estos casos).

En segundo lugar, sólo tenemos que asignar, desde el nuevo vector ordenado, las conferencias a las aulas, las asignará comprobando todas las aulas anteriores que tengan asignada una conferencia que se solapa con la que queremos asignar y encontrando el primer aula que en él no haya una conferencia que se solapa con la que queremos asignar. En caso de que no exista un aula donde asignar una conferencia porque ya existen otras conferencias asignadas o bien porque no se ha creado ningún aula. Dispondrá (creará) de otra aula, para poder hacer su asignación.

1.3. Elementos del Algoritmo

Ahora, identificamos a los elementos del algoritmo.

Conjunto de Candidatos (C) : Conjunto de todas las aulas creadas + 1 (una posible aula nueva que necesitemos crear).
Conjunto de Seleccionados (S): Conjunto de aulas asignadas.
Solución: Todas las conferencias estén asignadas a alguna aula.
Criterio de Factibilidad: que exista el número de aulas óptimo.
Criterio de Selección: la conferencia que tiene una menor hora de inicio Si.
Objetivo: el número de aulas debe ser el menor posible.

1.4. Demostración e indicios de la optimalidad del algoritmo

Nuestro algoritmo por su funcionamiento cumple las siguientes premisas:

Premisa 1) Dos conferencias que se solapan no pueden estar asignadas en el mismo aula.
Premisa 2) Las conferencias se asignan en la primera aula que esté libre desde el aula 1 hasta el aula n.
Premisa 3) Las conferencias se asignan ordenadas por menor hora de inicio Si.

Llamamos a, siendo a un número natural, a un número optimal de aulas que existe dado un vector de conferencias que cumple

$$a = max(Ao, \dots , An-1)$$

siendo Ao, ..., An-1 los subconjuntos de conferencias que se solapan entre sí y están en diferentes aulas.

Llamamos b al número natural de aulas que arroja el algoritmo dado un vector de conferencias.

Por teoría de números, b tiene que estar en alguno de estos 3 casos: b<a, b=a ó b>a. Ahora intentamos reducir a una contradicción los posibles casos.

Supongamos que b<a, esto implica:
1. Deben **existir conferencias que se solapan asignadas a la misma aula**. Ya que b sería menor que el máximo de los subconjuntos de conferencias que se solapan entre sí. Lo cual llevaría a una contradicción con la Premisa 1.

Supongamos que b>a, esto implica que debe **existir al menos una conferencia asignada en alguna de las aulas que van de c hasta n donde c=b-a**, para que tal asignación exista por tanto se debe cumplir uno de estos dos casos:

1. Debe haberse **asignado una conferencia A a un aula d+Z siendo Z un número natural entre d y n existiendo previamente un aula d que no tiene asignada una conferencia que se solape con esa.** Lo cual lleva a una contradicción con la Premisa 2.
2. Debe haberse **asignado una conferencia antes que otra que tuviera una hora de inicio Si menor.** Lo cual llevaría a una contradicción con la Premisa 3.

Por tanto, el único caso que nos queda es aceptar que b=a. Indicando la optimalidad del algoritmo.

2. Segunda Parte: Conferencia de presidentes

2.1. Enunciado formal del problema.

Se va a celebrar una conferencia de presidentes a la que asistirán n personas. Todas se van a sentar alrededor de una única gran mesa rectangular, como la de la figura, de forma que cada persona tendrá sentadas junto a él a otras dos personas (una a su izquierda y otra a su derecha). En función de las características de cada persona (por ejemplo categoría o puesto, lugar de procedencia,...) existen unas normas de protocolo que indican el grado de conveniencia de que dos personas se sienten en lugares contiguos (supondremos que dicho grado es un n´umero entero entre 0 y 100). El grado de conveniencia total de una asignaci´on de personas a su puesto en la mesa es la suma de todos los grados de conveniencia de cada persona con cada una de los dos personas sentadas a su lado. Se desea sentar a las personas de forma que el grado de conveniencia global sea lo mayor posible. Los asientos de la mesa los numeraremos como 0, 1, 2, hasta n−1, entendiendo que números contiguos representan asientos contiguos y que las posiciones 0 y n − 1 también son contiguas. También representaremos a cada persona como un número entre 0 y n − 1. Una instancia del problema ser´a una matriz c, de tama˜no n × n, de enteros entre 0 y 100, donde c[i][j] contiene el grado de conveniencia de sentar juntas a las personas i y j. No es necesario que la matriz sea simétrica, aunque lo supondremos por simplicidad. Una posible solución del problema se modeliza como un vector de enteros, a, de tamaño n, donde a[i] es la persona que es asignada al asiento i-ésimo. Obsérvese que cada posible solución del problema es una permutación de los enteros de 0 a n − 1 (si i 6= j entonces a[i] 6= a[j], una misma persona no se puede sentar en dos asientos distintos).

2.2. Implementación de algoritmos en C++.

Para la resolución de este ejercicio hemos diseñado dos algoritmos voraces.

2.2.1. Algoritmo 1

El primero de ellos se basa en crear una lista de candidatos para cada presidente y escoger el que tenga mayor afinidad (máximos locales). Los elementos del algoritmo son:
Conjunto de Candidatos (C) : Conjunto de presidentes afines a cada presidente.
Conjunto de Seleccionados (S): El vector asistentes *a*.
Solución: la conveniencia total de los seleccionados.
Criterio de Factibilidad: todos los presidentes están sentados
Criterio de Selección: el presidente con mayor conveniencia
Objetivo: la conveniencia total debe ser máxima

Para ello se ha hecho lo siguiente:

1. Se ordena el set de candidatos de cada presidente de mayor a menor afinidad. Es por eso que usamos la estructura set ya que nos hace la ordenación automática, solo tenemos que implementar el operador indicando el orden que queramos:
   ```
   bool operator()(const candidato & candidato1, const candidato &candidato2) const
   {
       return candidato1.valor > candidato2.valor;
   }
   ```
2. Tras ordenar, extraemos el primer elemento de cada set de *candidatos* (el mayor valor) y comprobamos que ese valor no pertenezca a un presidente ya introducido en el vector de asistentes *a*. Para ello hemos utilizado el método find de la stl:
   ```
   return find(a.begin(), a.end(), presidente) != a.end();
   ```

El algoritmo de greedy completo:

```
1. void algoritmo1(){
2.     for (int i=0; i<n; i++)
3.         a[i]=-1;
4.
5.     vector<set<candidato, comparaCandidatos>> candidatos;
6.     candidatos.resize(n);
7.
8.     //Ordena candidatos de mayor a menor empezando del
   presidente 0 a n-1
9.     int j = 0;
10.       for(auto it = candidatos.begin(); it !=
   candidatos.end(); ++it){
11.           for(int i = 0; i < n; i++){
12.               candidato nuevo;
13.               nuevo.presidente = i;
14.               nuevo.valor = c[i][j];
15.               (*it).insert(nuevo);
16.           }
17.           j++;
18.       }
19.
20.       //Mete al mejor candidato
21.       j=0;
22.       for(auto it = candidatos.begin(); it !=
   candidatos.end(); ++it){
23.           auto itset = (*it).begin();
24.           while( estaEnVector((*itset).presidente) and itset
   != (*it).end() ){
```

```
25.                 itset++;
26.             }
27.             a[j] = (*itset).presidente;
28.             j++;
29.         }
30.     }
```

Eficiencia Teórica

Las partes de código que presentan complejidad son:

```
31.     for (int i=0; i<n; i++)
32.             a[i]=-1;
```

Bucle for: O(n)

```
for(auto it = candidatos.begin(); it != candidatos.end();
++it){
```

Dos bucles for anidados: O(n²)

```
33.             for(int i = 0; i < n; i++){
34.                 candidato nuevo;
35.                 nuevo.presidente = i;
36.                 nuevo.valor = c[i][j];
37.                 (*it).insert(nuevo);
38.             }
39.             j++;
40.         }
```

```
41.   for(auto it = candidatos.begin(); it != candidatos.end();
      ++it){
```

Tres bucles anidados O(n³)

```
42.             auto itset = (*it).begin();
43.             while( estaEnVector((*itset).presidente) and itset
      != (*it).end() ){
44.                 itset++;
45.             }
46.             a[j] = (*itset).presidente;
47.             j++;
48.         }
```

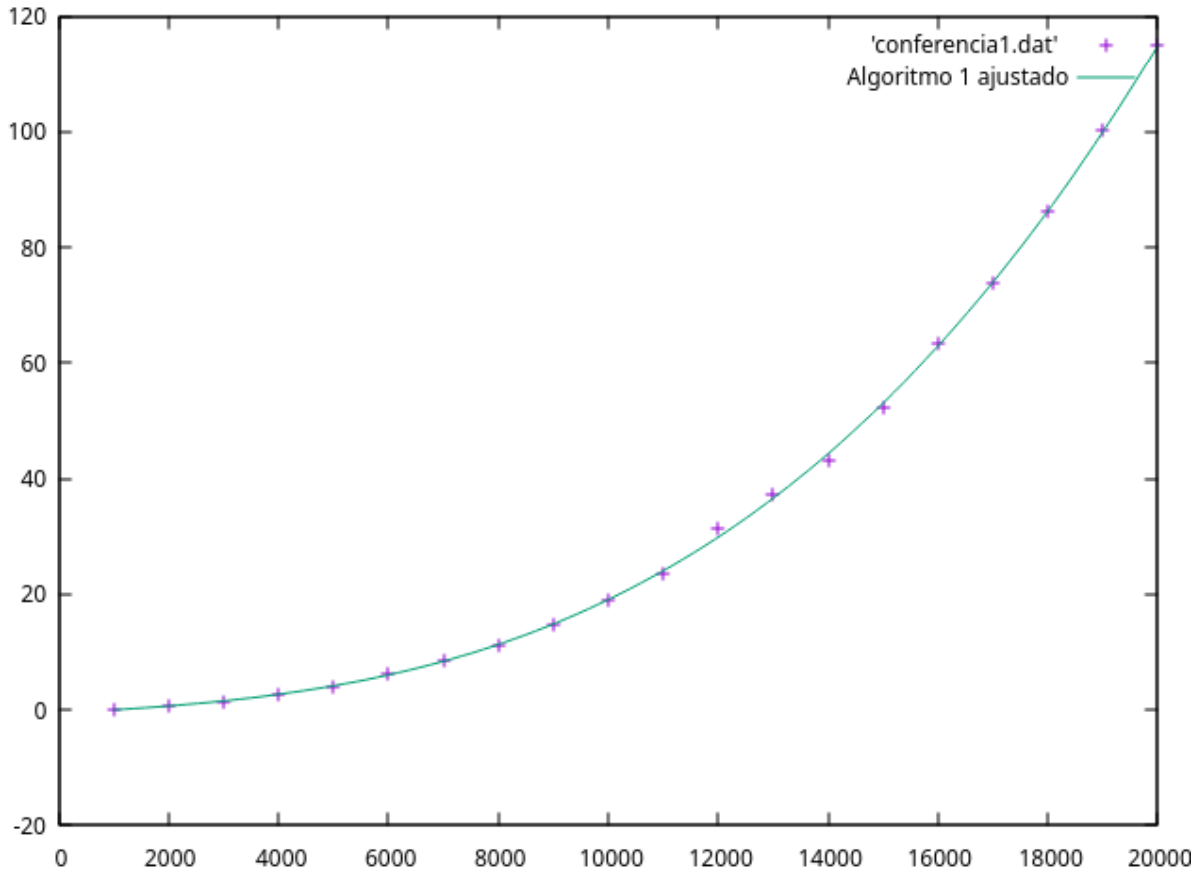
A parte, vemos que en estaEnVector() existe un bucle de complejidad O(n logn). Finalmente, este trozo de código es O(n^3) debido n (de la función estaEnVector) * n (del bucle while) * n (del bucle for).

Concluimos del análisis, que la complejidad del algoritmo es la mayor de los 3 bucles que hemos analizado por lo que sería O(n³).

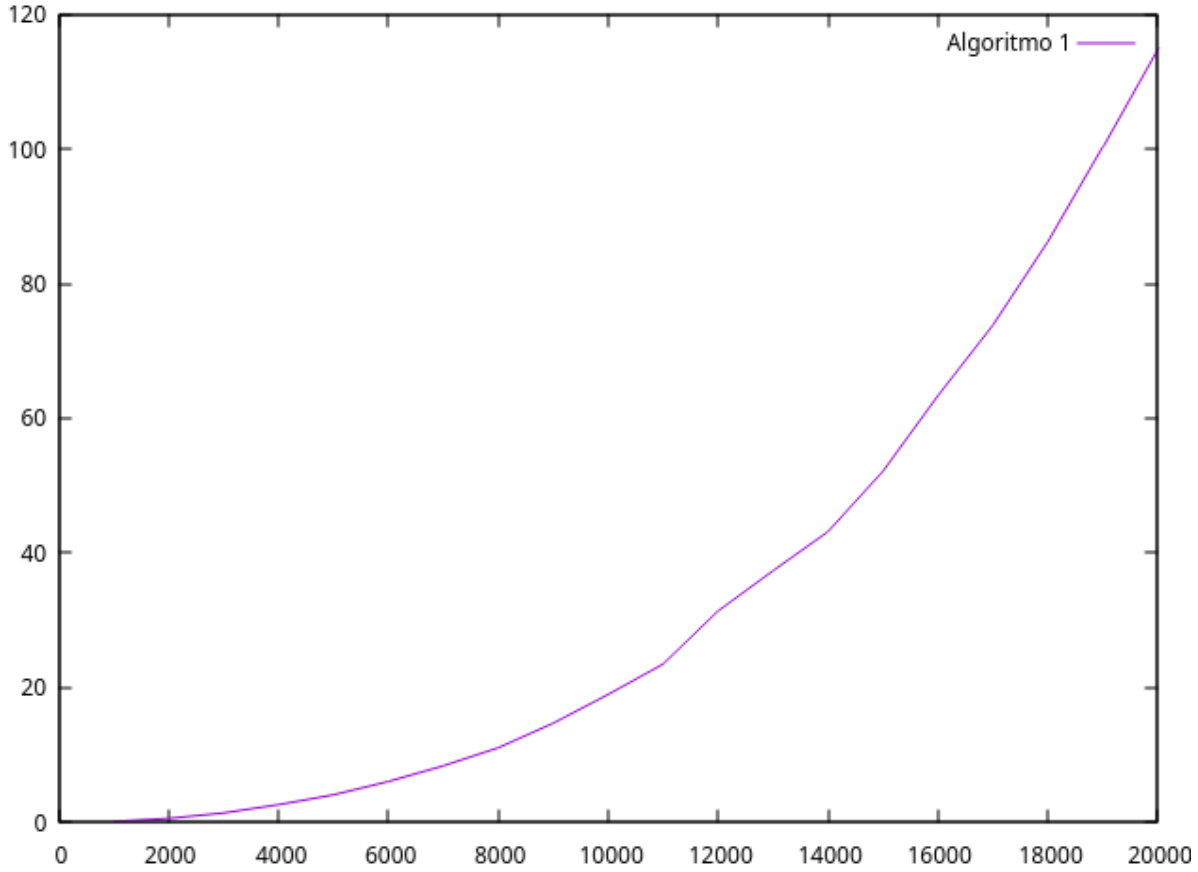
- Eficiencia Híbrida

```
iter      chisq       delta/lim  lambda   a0            a1            a2
a3

  0 1.5249824894e+13   0.00e+00  7.06e+05    3.993275e-07  -2.696588e-03
5.738027e+00   1
.000000e+00
  1 7.8286341292e+10  -1.94e+07  7.06e+04    1.434548e-07  -2.799701e-03
5.923507e+00   1
.000001e+00
  2 1.1668264331e+09  -6.61e+06  7.06e+03    3.479525e-08  -9.677791e-04
6.103597e+00   9
.999933e-01
  3 3.3026318743e+06  -3.52e+07  7.06e+02    1.737787e-09  -5.017852e-05
3.402866e-01   9
.999249e-01
  4 8.9780477791e+00  -3.68e+10  7.06e+01    1.128638e-11   5.194442e-08
1.478234e-04   9
.999183e-01
  5 7.8443395490e+00  -1.45e+04  7.06e+00    1.027557e-11   8.136068e-08
-5.143288e-05   9
.996568e-01
  6 7.7779147342e+00  -8.54e+02  7.06e-01    1.030150e-11   8.044920e-08
-4.207670e-05   9
.739484e-01
  7 6.0918740431e+00  -2.77e+04  7.06e-02    1.126880e-11   4.645557e-08
3.068417e-04   1
.532792e-02
  8 5.8195958303e+00  -4.68e+03  7.06e-03    1.184055e-11   2.636257e-08
5.130809e-04  -5
.512948e-01
  9 5.8195863172e+00  -1.63e-01  7.06e-04    1.184395e-11   2.624310e-08
5.143071e-04  -5
.546637e-01
iter      chisq       delta/lim  lambda   a0            a1            a2
a3


After 9 iterations the fit converged.
final sum of squares of residuals : 5.81959
rel. change during last iteration : -1.63467e-06

degrees of freedom    (FIT_NDF)                        : 16
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.603095
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.363724

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.1844e-11       +/- 9.079e-13    (7.665%)
a1              = 2.62431e-08      +/- 2.896e-08    (110.3%)
a2              = 0.000514307      +/- 0.0002651    (51.54%)
```

```
a3              = -0.554664        +/- 0.6588       (118.8%)

correlation matrix of the fit parameters:
               a0     a1     a2     a3
a0              1.000
a1             -0.988  1.000
a2              0.929 -0.974  1.000
a3             -0.732  0.807 -0.905  1.000
```



- Eficiencia Empírica



2.2.2. Algoritmo 2

En el segundo algoritmo inicialmente se selecciona a uno de los presidentes cuya conveniencia sea la máxima de todas.
Tras esto se irán seleccionando los presidentes que tengan mayor conveniencia con el presidente del extremo izquierdo o derecho de los presidentes ya seleccionados y se insertará a la izquierda o derecha según corresponda de modo que los presidentes se van seleccionando de forma adyacente.
Cuando todos los presidentes hayan sido seleccionados el algoritmo habrá finalizado.

```
void solucionGreedy(int n, int * a) {
      bool candidatos[n];
    int max = -1, pres, izq= n-1, der=1;
    bool insertarIzq;
    map<int, bool> sentados;

    // Función selección:
    // Inicialmente se selecciona a uno de los presidentes
    // que tienen la mejor conveniencia entre sí

    for(int i = 0; i < n; i++){
```

```
        for(int j = i+1; j < n; j++){
            if (c[i][j] > max){
                  max = c[i][j];
                  pres = i;
                  a[0]=i;
            }
        }
    }
    max = -1;
    a[0]=pres;
    sentados[pres]=true;
    // Función solución:
    // Cuando las posiciones a la izquierda y a la derecha de los
    // presidentes adyacentes se cruzan el vector con la solución
    // estará lleno
    while (der != (izq + 1) % n){
      max = -1;
      for (int i = 0; i < n; i++){

            // Se selecciona al candidato que mejor conveniencia tenga
            // con los presidentes de la izquierda o de la derecha
            if (c[a[(izq+1) % n]][i] > max and sentados[i] == false){
                  max = c[a[(izq+1) % n]][i];
                  pres = i;
                  insertarIzq=true;
            }
            if (c[a[(der+n-1)%n]][i] > max and sentados[i] == false){
                  max = c[a[(der+n-1)%n]][i];
                  pres = i;
                  insertarIzq=false;
            }
        }

        // Se elimina al seleccionado de los candidatos
        sentados[pres]=true;
        // Se inserta al presidente a la izquierda o a la derecha
        // de los presidentes ya seleccionados de forma adyacente
        if (insertarIzq){
            a[izq]=pres;
            izq = (izq + n-1) % n;
        }else{
            a[der]=pres;
            der = (der + 1) % n;
        }
    }
}
```

Eficiencia Teórica

Las dos únicas partes del código que presentan complejidad son:

Itera n-1 veces (Hasta que izq y der se cruzan)

```
while (der != (izq + 1) % n){                          Bucle while: O(n²)
      max = -1;
            Itera n veces
      for (int i = 0; i < n; i++){
       ..........
      O(1)
      }
        if (insertarIzq){
            a[izq]=pres;
                                          izq disminuye
            izq = (izq + n-1) % n;
        }else{
            a[der]=pres;
                                          izq aumenta
            der = (der + 1) % n;
        }
}
                    Itera n veces
for(int i = 0; i < n; i++){                    Bucle for externo: ≤O(n²)
                    Itera menos de n veces
        for(int j = i+1; j < n; j++){
            O(1)
            if (c[i][j] > max){
                  max = c[i][j];
                  pres = i;
                  a[0]=i;
            }
        }
    }
```

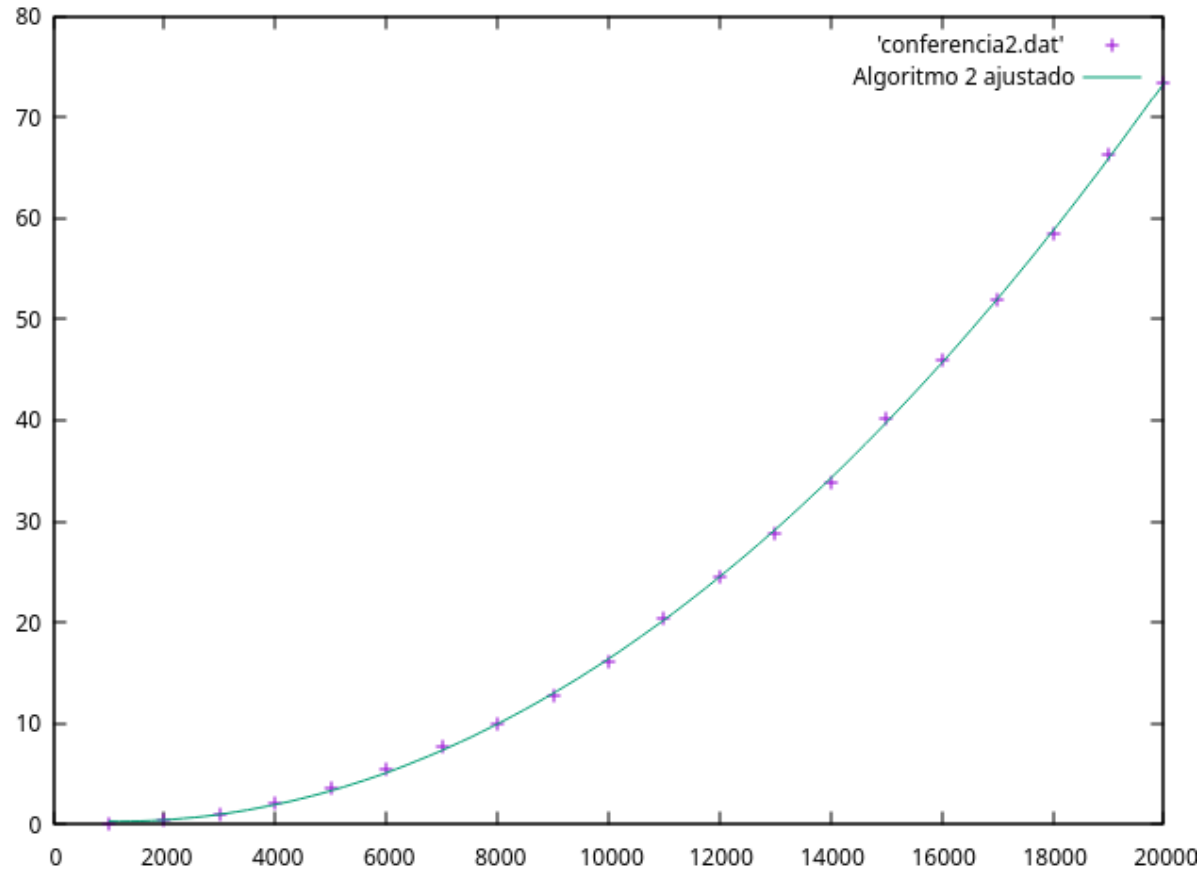
Del análisis de los bucles podemos concluir que el bucle while es O(n²) y el bucle for es menor o igual a O(n²) y ,dado que el resto son sentencias constantes, el algoritmo sería cuadrático (O(n²))

Eficiencia Híbrida

Al hacer el ajuste a un polinomio cuadrático, nos arroja este ajuste.

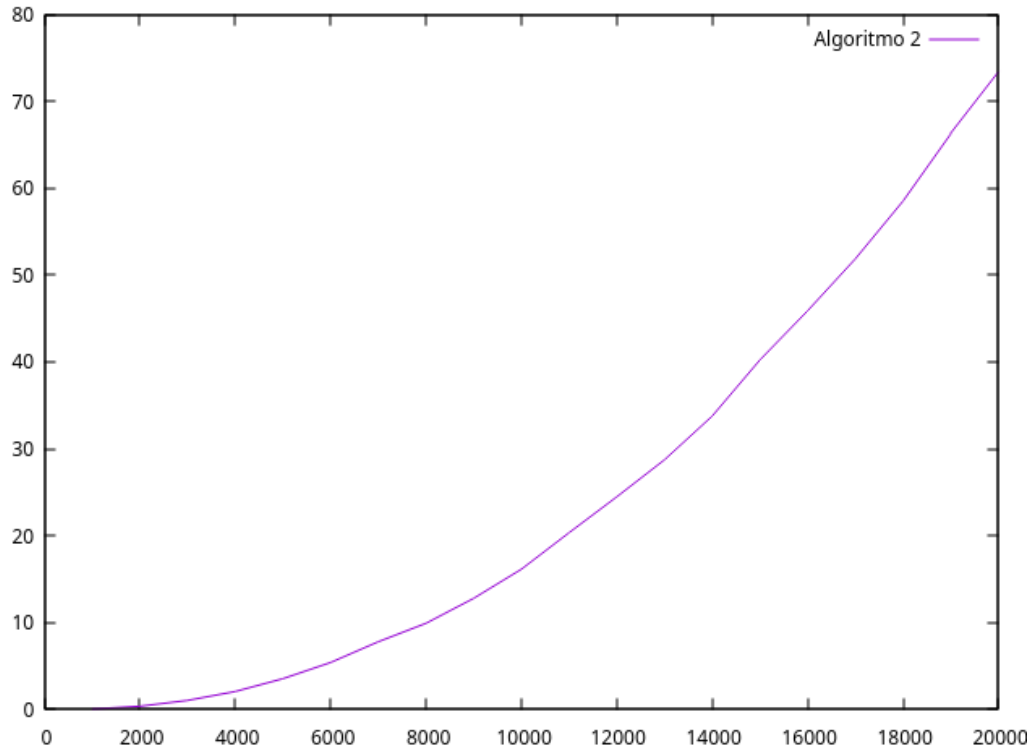
```
iter       chisq       delta/lim  lambda   a0             a1             a2
  0 7.2275394970e+17   0.00e+00  1.10e+08    1.000000e+00   1.000000e+00
1.000000e+00
  1 1.9423666111e+14  -3.72e+08  1.10e+07    1.633359e-02   9.999400e-01
1.000000e+00
  2 1.8441045280e+08  -1.05e+11  1.10e+06   -5.811312e-05   9.999375e-01
1.000000e+00
  3 1.7896364282e+08  -3.04e+03  1.10e+05   -6.083601e-05   9.997889e-01
1.000000e+00
  4 1.7376508809e+08  -2.99e+03  1.10e+04   -5.994285e-05   9.851528e-01
9.999971e-01
```

```
  5 2.8142843878e+07  -5.17e+05  1.10e+03   -2.399850e-05   3.961333e-01
9.998823e-01
  6 1.2611861200e+03  -2.23e+09  1.10e+02    4.721812e-08   2.096456e-03
9.998049e-01
  7 1.7319373057e+00  -7.27e+07  1.10e+01    2.091490e-07  -5.570980e-04
9.997451e-01
  8 1.7236456831e+00  -4.81e+02  1.10e+00    2.091136e-07  -5.561377e-04
9.939020e-01
  9 1.4933663750e+00  -1.54e+04  1.10e-01    2.072616e-07  -5.105785e-04
7.601794e-01
 10 1.4495052621e+00  -3.03e+03  1.10e-02    2.060351e-07  -4.804066e-04
6.053955e-01
 11 1.4495033384e+00  -1.33e-01  1.10e-03    2.060269e-07  -4.802055e-04
6.043637e-01
iter      chisq       delta/lim  lambda   a0            a1            a2

After 11 iterations the fit converged.
final sum of squares of residuals : 1.4495
rel. change during last iteration : -1.32718e-06

degrees of freedom    (FIT_NDF)                        : 17
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.292002
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0852649

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 2.06027e-07      +/- 2.204e-09    (1.07%)
a1              = -0.000480205     +/- 4.764e-05    (9.922%)
a2              = 0.604364         +/- 0.2172       (35.95%)

correlation matrix of the fit parameters:
               a0     a1     a2
a0              1.000
a1             -0.971  1.000
a2              0.781 -0.889  1.000
```



Eficiencia Empírica

Ahora mostramos la gráfica de la que representa la eficiencia empírica.



2.2.3. Comparativa de eficiencia en los dos algoritmos.

A continuación, mostramos una gráfica de las eficiencias de los dos algoritmos implementados.