UNIVERSIDAD
DE GRANADA

E.T.S. Ingeniería Informática y Telecomunicaciones

Doble Grado en Ingeniería Informática y Matemáticas

ALGORÍTMICA

Práctica 3
Algoritmos Voraces

Diego Ortega Sánchez
e-mail: diegoos03@correo.ugr.es

Gonzalo Martínez Piédrola
e-mail: gonzalomp@correo.ugr.es

Jorge García Gámiz
e-mail: jorgegarciag@correo.ugr.es

Curso 2022-2023

Índice

1. Autores. **1**

2. Objetivos **2**

3. Definición del problema **3**
- 3.1. Servicio de catering . . . 3
- 3.2. El viajante de comercio (PVC) . . . 6
- 3.3. Descripción del entorno . . . 8

4. Algoritmo diseñado **10**
- 4.1. Demostración de su validez . . . 15
- 4.2. Análisis de eficiencia: teórica, empírica e híbrida . . . 15
  - 4.2.1. Ajustes con regresión . . . 19

5. Problema del Viajante de comercio **24**
- 5.1. Primera heurística . . . 24
  - 5.1.1. Justificación de su validez (PVC1) . . . 27
- 5.2. Segunda heurística . . . 27
  - 5.2.1. Justificación de su validez (PVC2) . . . 32
- 5.3. Tercera heurística . . . 33
  - 5.3.1. Justificación de su validez (PVC3) . . . 42
- 5.4. Análisis comparativo de las tres heurísticas . . . 42

6. Conclusiones **45**

1. Autores.

Este es el grupo Negreira Boys. El problema con el que nos ha tocado trabajar ha sido “Servicio de catering”.

A lo largo de la realización del trabajo, todos hemos trabajado en todas las partes del proyecto. Sin embargo, como es natural, cada uno se ha centrado de manera más ahonda en una parte específica, las cuales se comentan a continuación. El reparto del trabajo se ha intentado hacer de la forma más equitativa posible, siempre procurando que cada integrante participase en cada una de las distintas facetas del proyecto.

Los integrantes del grupo somos:

- **Diego Ortega Sánchez**: $33,\widehat{3}\,\%$

  Principalmente, ha implementado los códigos, tanto el específico como el dyv, estudiando los análisis También ha contribuido con la redacción de partes de la memoria relativas a dichos algoritmos y la sección de análisis.

- **Gonzalo Martínez Piédrola**: $33,\widehat{3}\,\%$

  Se ha centrado en el cálculo de umbrales (teórico, óptimo y de tanteo) y en los análisis (especialmente en el teórico). También contribuyó en la implementación de los códigos. Asimismo, se encargó de la redacción de la parte de umbrales.

- **Jorge García Gámiz**: $33,\widehat{3}\,\%$

  Primordialmente, se ha encargado de la memoria y de la presentación. A su vez, también ha asumido gran parte del análisis híbrido y empírico de los dos algoritmos. Por último, contribuyó en la depuración del algoritmo.

2. Objetivos

El objetivo de esta práctica consiste, primordialmente, en asimilar el funcionamiento de los algoritmos llamados “voraces” o “greedy”. Sabemos, por lo visto en teoría, que dicho se trata de un algoritmo que encuentra una solución globalmente óptima a un problema a base de hacer elecciones localmente óptimas. Es decir: el algoritmo siempre hace lo que “parece” mejor en cada momento, sin tener nunca que reconsiderar sus decisiones, y acaba llegando directamente a la mejor solución posible.

Con este fin, se ha propuesto un problema a cada grupo y uno para todos. Estos deben ser resueltos teniendo en cuenta la implementación de los ya mencionados algoritmos “voraces”. De tal modo, estudiaremos la eficiencia de ambos algoritmos, tanto a nivel teórico como empírico e híbrido.

Los objetivos del estudio son:

1. Implementación correcta de los algoritmos.
2. Aplicación de lo aprendido en la Práctica 1 a la hora de realizar el análisis híbrido y empírico.
3. Cálculo acertado del análisis teórico de todos los algoritmos, con la correcta resolución de la recurrencia planteada.
4. Correctas interpretaciones de la información obtenida.
5. Capacidad para aminorar y plasmar en la memoria el aprendizaje y resultado del estudio.

3. Definición del problema

3.1. Servicio de catering

Se considera el siguiente problema:

Servicio de catering

- Una empresa de catering debe servir a los n comensales de un banquete para lo que dispone de un total de c camareros, con $c < n$.
- Los comensales se encuentran sentados en mesas individuales, distribuidas por el local.
- Cada camarero sólo puede atender a un comensal a la vez. Cuando todos los camareros están ocupados atendiendo a comensales, el resto de comensales deberán esperar a que alguno de los camareros quede libre.
- Dado que un camarero puede llevar solamente un producto a la vez, el tiempo de espera de cada comensal dependerá de:
    1. lo lejos que esté su mesa de la salida de cocina (desplazamientos)
    2. de los comensales que el camarero que le sirve la comida haya tenido que atender con anterioridad.
- Claramente, cuanto mayor es el tiempo de espera de un comensal mayor será el grado de insatisfacción con el servicio.

Podemos generar ficheros de este tipo arbitrariamente grandes con el programa *data_generator.cpp*, que recibe un número entero n y una cadena de caracteres r. El programa crea un archivo llamado r con n puntos generados aleatoriamente. El programa en cuestión lo implementamos en esta memoria a continuación:

```
#include <iostream>
#include <random>
#include <fstream>
#include <cstdlib>
#include <ctime>

using namespace std;

```

```
int main(int argc, char *argv[])
{
        // Verificar que se haya pasado un argumento para el número de líneas
        if (argc != 3)
        {
                cout << "Uso: " << argv[0] << " <numero de lineas> " << "<nombre del fichero>" << endl;
                return 1;
        }

        // Convertir el argumento a un entero
        int num_lines = atoi(argv[1]);
        string file_name = argv[2];

        // Inicializar la semilla aleatoria con la hora actual
        srand(time(nullptr));

        // Abrir el archivo para escribir
        ofstream file(file_name, ios::trunc);

        // Generar el número de líneas especificado con dos números aleatorios tipo float en cada línea
        for (int i = 0; i < num_lines; i++)
        {
                std::random_device rd; // Genera una semilla aleatoria
                std::mt19937 gen(rd()); // Generador de números aleatorios
                std::uniform_int_distribution<> dis(1, 10000); // Distribución uniforme entre -100 y 100

                int num1 = dis(gen);

                if(i == num_lines - 1){
                        file << num1;
                }
                else{
                        file << num1 << "\n";
                }
```

```
        }

        // Cerrar el archivo
        file.close();

        return 0;
}
```

La medida de tiempos se realiza en los propios códigos implementados. Se mide el tiempo antes y después de ejecutar el respectivo algoritmo y se calcula el tiempo transcurrido. Para ello se ha decidido utilizar la biblioteca chrono, en concreto, la opción más precisa (*high_resolution_clock*).

```
clock_t t_antes;
clock_t t_despues;

//Captura el valor del reloj antes de la llamada a burbuja
t_antes = clock();

// Llama al algoritmo de ordenación <nombre_algoritmo>
nombre_algoritmo(array, 0, n);

//Captura el valor del reloj después de la ejecución del
   código
t_despues = clock();

// Para obtener el número de segundos el cálculo es
   sencillo:
cout << "n" << "\t";
cout << (double)(t_despues - t_antes) / CLOCKS_PER_SEC <<
   endl;
```

A nivel práctico hemos usado el siguiente ejecutable para, tal como indica su nombre, medir los tiempos. Se trata del archivo *mide-tiempos.sh*, cuyo contenido mostramos a continuación:

```
#!/bin/bash
#echo "" >> salida.dat
i=100000
while [ "$i" -le 2500000 ]
do
        ./Generador/datagen $i $3
        ./$1 $2 $3 /dev/null >> $1TablaDatos$2.dat
        i=$(( $i + 100000 ))
done
```

Nota: *los datos que aparecen son unos datos que ejemplifican una de las alternativas para las distintas ejecuciones y pruebas que hemos tenido que realizar. Más allá de eso, no son más significativas.*

3.2. El viajante de comercio (PVC)

Por otro lado, deberemos tratar también con el siguiente problema, que describimos a continuación:

El viajante de comercio (PVC)

Una empresa que comercializa componentes informáticos ha contratado un nuevo comercial. Al agente le ha asignado un conjunto de clientes, facilitándole la información relevante de cada uno, que incluye nombre, correo electrónico, teléfono y dirección postal. El director comercial le encarga al nuevo agente visitar a todos esos clientes, con el objetivo de hacer buenas ventas y que no vuelva hasta que haya visitado a todos los clientes. Por tanto, el recorrido considerado debe comenzar y finalizar en la empresa. Además, sólo podrá visitar cada cliente una sola vez. Además, para reducir gastos de desplazamiento, la distancia total recorrida debe ser lo más corta posible.

Al igual que ocurría para el problema previo, podemos generar ficheros de este tipo arbitrariamente grandes con un‘ programa *data_generator.cpp*. Huelga decir que este será el generador empleado para las tres heurísticas. Lo implementamos en esta memoria como sigue:

```
#include <iostream>
#include <fstream>
#include <random>
#include <set>

using namespace std;

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

        if (argc != 2) {
                std::cout << "Uso: " << argv[0] << " <
                   número de líneas>\n";
                return 1;
```

```
        }
        int n = std::stoi(argv[1]);

        // Crear un generador de números aleatorios
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<int> dis(-1000, 1000)
           ;

        // Crear un conjunto para evitar duplicados
        set<pair<int, int>> lines;

        // Generar n líneas aleatorias únicas
        while (lines.size() < n+1) {
                int a = dis(gen);
                int b = dis(gen);
                lines.insert({a, b});
        }

        // Escribir las líneas en el archivo
        ofstream file("Generador-pvc/data");
        for (const auto& [a, b] : lines) {
                file << a << " " << b << "\n";
        }
        file.close();

        return 0;
}
```

En este caso, vamos a estudiar el problema implementado distintas heurísticas, las cuales pasaremos posteriormente a comparar por medio de un análisis como los que ya hemos realizado en repetidas ocasiones a lo largo de la asignatura.

La medida de tiempos se realiza en los propios códigos implementados. Se mide el tiempo antes y después de ejecutar el respectivo algoritmo y se calcula el tiempo transcurrido. Para ello se ha decidido utilizar la biblioteca chrono, en concreto, la opción más precisa (*high_resolution_clock*).

```
clock_t t_antes;
clock_t t_despues;

//Captura el valor del reloj antes de la llamada a burbuja
t_antes = clock();
```

```
// Llama al algoritmo de ordenación <nombre_algoritmo>
nombre_algoritmo(array, 0, n);

//Captura el valor del reloj después de la ejecución del
   código
t_despues = clock();

// Para obtener el número de segundos el cálculo es
   sencillo:
cout << "n" << "\t";
cout << (double)(t_despues - t_antes) / CLOCKS_PER_SEC <<
   endl;
```

A nivel práctico hemos usado el siguiente ejecutable para, tal como indica su nombre, medir los tiempos. Se trata del archivo *mide-tiempos_pvc.sh*, cuyo contenido mostramos a continuación:

```
#!/bin/bash
#echo "" >> salida.dat
i=1000
while [ "$i" -le 25000 ]
do
        ./Generador-pvc/datagen $i
        ./$1 $2 >> $1TablaDatos.dat
        i=$(( $i + 1000 ))
done
```

***Nota**: los datos que aparecen son unos datos que ejemplifican una de las alternativas para las distintas ejecuciones y pruebas que hemos tenido que realizar. Más allá de eso, no son más significativas.*

3.3. Descripción del entorno

Para dar por concluida esta parte de la sección, mencionamos algunas especificaciones que pueden afectar en el tiempo de ejecución de los algoritmos. Para ello, vamos a hablar, brevemente, del entorno de trabajo. Mostramos algunas características del hardware y software con al orden neofetch (véase figura 3.3). El modelo del hardware se trata de HP HP Pavilion Laptop 15-cs0xxx, con un procesador $Intel^{®}$ $Core^{TM}$ sf i7-8550U CPU @ 1,80$GHz \times 8$ y una memoria de 8GB. Hemos trabajado en el kernel Linux (versión 5.19.0-35-generic), con el sistema operativo (SO) de Ubuntu 22.04.1 LTS x86_64, tipo 64 bits y versión de GNOME 42.2. Por su parte, el shell de bash es la versión 5.1.16.

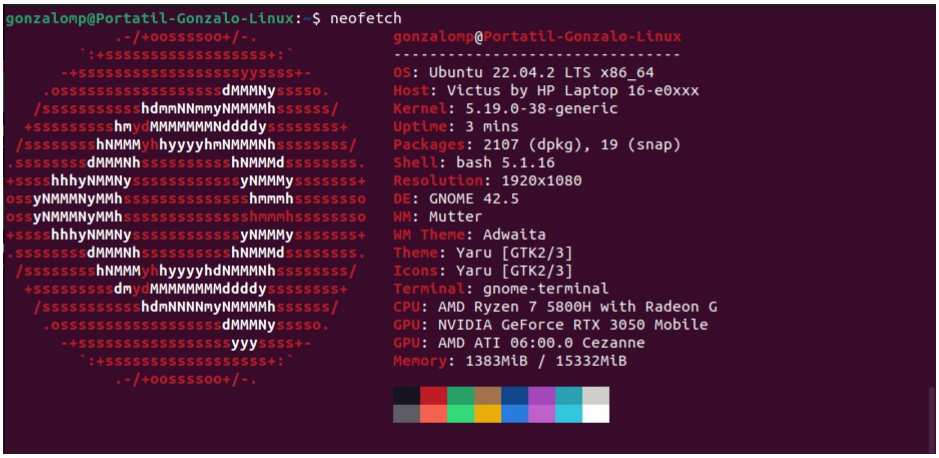

Figura 1: Comando neofetch en el ordenador en el que hemos ejecutado los programas.

En cuanto a la implementación, los programas se han escrito en lenguaje C++ en el editor Visual Studio Code. También indicamos la opción de compilación, que es

```
g++ -g <archivo.cpp> -o <ejecutable> -std=gnu++0x
```

Siendo la información más específica del SO y del compilador la que sigue:

```
g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
```

Como podemos comprobar, hemos decidido compilar los programas sin ningún tipo de optimización explícita, es decir, usando la optimización -O2.

Por último, por comentar brevemente algo sobre el editor de textos que hemos escogido para redactar la memoria, este se trata de LaTeX, con entorno de TeXstudio. Sin embargo, para hacer las tablas, hemos decidido implementarlas desde un programa externo llamado R studio, donde nos es más sencillo trabajar con este tipo de datos.

4. Algoritmo diseñado

En el siguiente cuadro resumen, vamos a plasmar la idea que hemos seguido a la hora de la implementación del algoritmo que hemos utilizado.

Diseño del Algoritmo Voraz

- Candidatos: comensales $= A$. (En A almacenamos el tiempo que se tarda de cocina a cada comensal.)
- Seleccionados: todos los comensales se reparten entre c camareros con $c < A.size()$.

  $\star$ camarero i-ésimo: $S_i = \{\emptyset\}\ \forall i \in \{1,\ldots,c\}$, $M = \{S_i : i \in \{1,\ldots,c\}\}$
- Función selección: el mejor candidato será el comensal con tiempo desde cocina a mesa menor.
- Función factibilidad: $A[i]$ se añade a S_i para los primeros c comensales. Una vez haya terminado un camarero de atender a un comensal, busca al siguiente candidato, así sucesivamente hasta que todos los comensales hayan sido atendidos.
- Objetivo: minimizar el tiempo de servicio.
- Solución: asignación de comensales a camareros de manera que $t(S_j)$ sea lo menor posible sin que haya camareros ociosos durante mucho tiempo. En caso de A ordenado quedaríamos con

$$S_i = \{A[i],\ A[i+c],\ A[i+2c],\ldots,\ A[i+kc]\} \quad (k \in \mathbb{N}),$$

  obteniendo le mejor tiempo de servicio.

Mostramos a continuación el algoritmo diseñado para la resolución de nuestro ejercicio.

```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
#include <random>
#include <fstream>
#include <sstream>
```

```
#include <chrono>

using namespace std;
using namespace std::chrono;


/*
->Algoritmo Voraz encargado de resolver el Problema 1: Servicio de Catering

Recibe los siguientes parámetros:
-comensales (vector<float>): Un vector con los tiempos que tarda un camarero
en llegar a la mesa del comensal desde cocina.
-c (int): Indica el número total de camareros en servicio

Devuelve: Un vector de vectores de tipo float, donde cada vector representa
un camarero con los comensales que les han sido asignados.

*/
vector<vector<float>> VorazCatering (vector<float> comensales, int c){

        float totalTime = 0;

        for(int i = 0; i < comensales.size(); i++)
        totalTime += comensales[i];

        //Tiempo ideal que debería tardar cada camarero
        float maxTimeEach = totalTime / c;

        //Ordenamos el vector de comensales en orden creciente
        sort(comensales.begin(), comensales.end());

        vector<vector<float>> camareros; //Vector de camareros
        vector<float> camareroIesimo; //Vector de comensales asignados a un camarero

        //Asignamos a cada camarero los comensales que le corresponden
```

```
        // siempre y cuando el tiempo total que tarda el camarero en atenderlos
        // a todos no supere maxTimeEach
        for(int i = 0; i < c; i++){

                for(int j = i; j < comensales.size(); j += c){

                        camareroIesimo.push_back(comensales[j]);

                }

                camareros.push_back(camareroIesimo); // Añadimos el camarero
                camareroIesimo.clear(); //Borramos para inicializar un nuevo camarero


        }

        return camareros;
}

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

        // Comprobamos que se ha pasado un argumento con la ruta del fichero
        if (argc != 4) {
                cerr << "Uso: " << argv[0] << " <n_camareros> " << " <fichero_entrada> "
                << " <fichero_salida> " << endl;
                return 1;
        }

        vector<float> comensales;
        string line;
        int ncamareros = stoi(argv[1]); // Leemos número de camareros
        //cout << "Camareros -> " << ncamareros << endl;

        //ENTRADA DEL PROBLEMA

```

```
        // Abrimos el fichero
        ifstream file(argv[2]);
        if (!file.is_open()) {
                cerr << "No se pudo abrir el fichero" <<
                   endl;
                return 1;
        }

        // Leemos el fichero línea a línea
        while (getline(file, line)) {

                // Parseamos la línea para obtener el
                   tiempo del comensal
                istringstream iss(line);
                float comensal;
                if (!(iss >> comensal)) {
                        cerr << "Error al leer el fichero"
                           << endl;
                        return 1;
                }

                // Lo añadimos al vector
                comensales.push_back(comensal);
        }

        // Cerramos el fichero
        file.close();

        //Comprobamos si los datos recibidos son correctos
        if(!(ncamareros < comensales.size())){
                cerr << "Error en los datos: " << endl;
                cerr << "Numero camareros mayor al de
                   comensales " << endl;
                return 1;
        }

        //CÁLCULO DEL TIEMPO EMPÍRICO

        //Inicializamos las variables para calcular el
           tiempo que tarda el algoritmo
        high_resolution_clock::time_point t_antes,
           t_despues;
        duration<double> transcurrido;
```

```
    //Pasamos los datos del vector y calculamos el tiempo que tarda
    t_antes = high_resolution_clock::now();
    vector<vector<float>> sol = VorazCatering(comensales, ncamareros);
    t_despues = high_resolution_clock::now();

    transcurrido = duration_cast<duration<double>>(t_despues - t_antes);
    cout << comensales.size() << "\t" << transcurrido.count() << endl;


    //SALIDA DEL PROGRAMA (Solución al Problema)

    //Abrimos el fichero de salida
    ofstream ofile(argv[3]);

    if (!ofile.is_open()) {
            cerr << "No se pudo abrir el fichero de salida" << endl;
            return 1;
    }

    float time = 0, totalTime = 0;
    vector<float> allTimes;
    for(int i = 0; i < sol.size(); i++){

            for(int j = 0; j < sol[i].size(); j++)
            time += sol[i][j];

            line = to_string(i) + ": " + to_string(time) + " s\n";
            ofile << line;
            allTimes.push_back(time);
            totalTime += time;
            time = 0;
    }

    //cout << "Tiempo Maximo medio: " << totalTime/ncamareros << endl;

```

```
        //Cerramos el fichero
        ofile.close();


        return 0;
}
```

4.1. Demostración de su validez

Supongamos que $\exists C$ asignación tal que permite asignar n comensales a c camareros haciendo que el servicio se haga en menos tiempo. Sea r arbitrario el número de iteraciones tal que la asignación de comensales $C = A$, es decir:

1. Asignamos los primeros c comensales (uno a cada camarero).
2. Según termine un camarero se le asigna otro comensal, hasta que hayan sido asignados los c siguientes comensales.
3. Repetimos este proceso hasta que todos los comensales hayan sido atendidos.

$$r < k, \;\; A_i = C_i \; \forall i \in \{1, \ldots, r\}$$

Tenemos entonces que $A_{r+1} > C_{r+1}$, por como hemos planteado el problema, es decir, que en la asignación $(r+1)$-ésima de los siguientes c comensales la asignación C es más rápida, lo que hace que se atiendan c comensales antes. Pero esto es imposible, pues el algoritmo A escoge a los c comensales más cercanos a cocina disponibles (séanse, sin atender) en ese momento. Hemos llegado a un absurdo ($\Rightarrow\Leftarrow$), luego $r = k \implies A = C$.

4.2. Análisis de eficiencia: teórica, empírica e híbrida

En esta sección, vamos a ahondar en los resultados obtenidos para el algoritmo implementado.

Empezamos hablando de la eficiencia teórica. Si nos fijamos en el código, a primera vista pudiera parecer que la eficiencia del algoritmo es $\mathcal{O}(n^2)$ pues el ciclo “while” en el peor de los casos, se ejecutaría n veces, al igual que el ciclo “for”, que se repite n veces. Sin embargo, la ventaja de este algoritmo es que cada punto se recorre como máximo 2 veces durante toda la ejecución, una al comprobar si supone un giro a la izquierda respecto al siguiente punto, y otra en caso de encontrar un punto siguiente en el vector ordenado que suponga un giro a la derecha, al ir borrando los puntos. Así pues, tenemos que la eficiencia de este bucle no deja de ser $\mathcal{O}(n)$, es decir, lineal, lo que nos deja que la mayor parte del tiempo que emplea el algoritmo es en la ordenación $sort\_by\_orientation$ que como sabemos, tiene eficiencia $\mathcal{O}(n\log(n))$.

Una vez implementado el algoritmo, lo hemos ejecutado aplicándoselo a distintos vectores con tamaños distintos así como con un número distinto de camareros (1700, 50000 y 99999) y viendo los tiempos obtenidos. Los datos obtenidos de estos estudios pueden consultarse en la Tabla 4.2 y gráficamente en las imágenes 4.2, donde vemos una comparación de los tres casos de forma visual y en 4.2 donde vemos las gráficas de forma individual.

Nº Comensales	**1700** camareros	**50000** camareros	**99999** camareros
100000	0.02355	0.03165	0.04016
200000	0.04736	0.05518	0.06355
300000	0.07148	0.07948	0.09005
400000	0.09662	0.10495	0.11259
500000	0.12232	0.12935	0.14037
600000	0.14754	0.15530	0.16324
700000	0.17386	0.18265	0.19100
800000	0.19715	0.21125	0.21297
900000	0.22565	0.24396	0.23992
1000000	0.25112	0.26199	0.27060
1100000	0.27763	0.28778	0.29194
1200000	0.30645	0.31659	0.32149
1300000	0.35877	0.33969	0.34281
1400000	0.37020	0.36736	0.37296
1500000	0.38616	0.39597	0.39849
1600000	0.41489	0.42148	0.42968
1700000	0.44267	0.45264	0.45161
1800000	0.47185	0.48445	0.48195
1900000	0.49806	0.50252	0.50763
2000000	0.52366	0.53543	0.54979
2100000	0.55945	0.56443	0.56812
2200000	0.58665	0.59088	0.58996
2300000	0.60758	0.61631	0.63272
2400000	0.63814	0.65285	0.65771
2500000	0.67404	0.69578	0.67664

Cuadro 1: De izquierda a derecha: caso en el que hay 1700 camareros, caso en el que hay 50000 y caso en el que hay 99999. Los datos obtenidos representan el tiempo en segundos que se tarda para uno de los casos descritos.

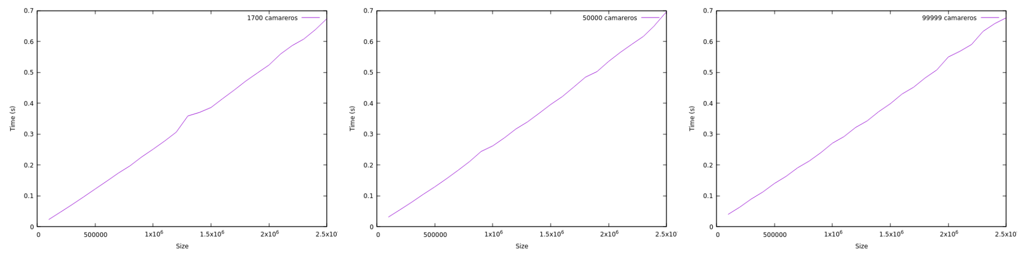

Figura 2: De izquierda a derecha: caso en el que hay 1700 camareros, caso en el que hay 50000 y caso en el que hay 99999.

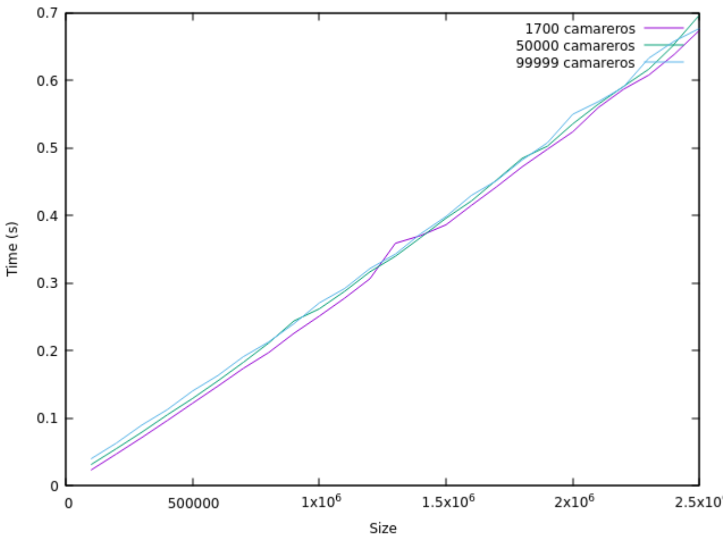

Figura 3: Comparación de los tres casos considerados para el problema en lo referente al número de camareros: 1700, 50000 y 99999.

La conclusión inmediata y evidente que sacamos de esta información una vez la analizamos, es que el número de camareros no afecta en el tiempo empleada para la asignación (se asignan el mismo número de comensales a cada camarero), lo cual es coherente. En cambio sí afecta en el tiempo que se tarda en lo referente al cumplimiento del servicio, como recogemos en la Tabla 4.2.

Número de comensales	Tiempo total
10000	250078.00
20000	124953.00
30000	83501.10
40000	62508.50
50000	49978.70
60000	41618.70
70000	35711.30
80000	31254.70
90000	27787.10
100000	24978.80
110000	22734.50
120000	20859.00
130000	19238.40
140000	17838.40
150000	16674.10
160000	15624.60
170000	14727.60
180000	13904.00
190000	13158.70
200000	12496.30
210000	11894.40
220000	11361.20
230000	10869.20
240000	10409.00
250000	9997.67

Cuadro 2: Tiempo que se emplea dependiendo del número de comensales para un número de camareros fijo, como ya hemos comentado, de 50000 camareros.

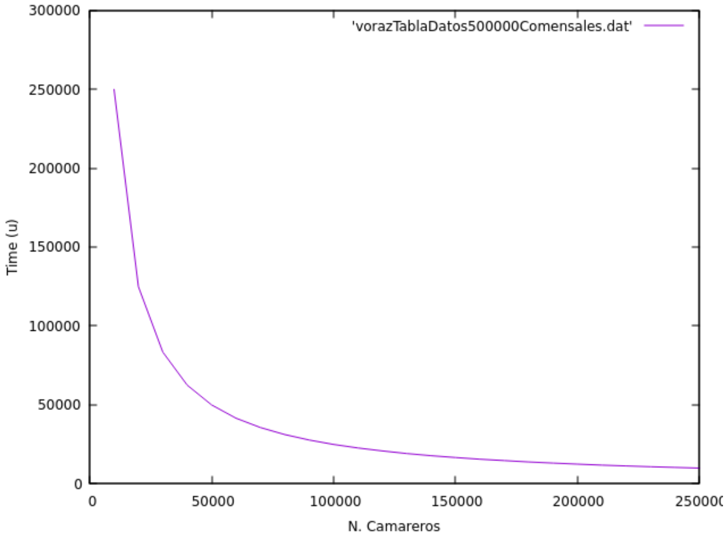

Figura 4: Tiempo que se emplea dependiendo del número de comensales para un número de camareros fijo, como ya hemos comentado, de 50000 camareros.

4.2.1. Ajustes con regresión

A continuación, vamos a presentar los distintos ajustes que hemos empleado para la representación de los datos y vamos a ver cómo de buenos son dichos ajustes.

Sin embargo, corresponde comentar que los únicos datos que hemos ajustado son los que hemos obtenido para el caso en el que se dispone de 50000 camareros. Esto se debe a que de otra manera se trataría de un documento más repetitivo y hemos de valorar el dinamismo de su lectura de cara a su entendimiento y máximo provecho.

Los ajustes empleados son los siguientes:

Ajustes empleados para el problema del catering

Lineal $\mathcal{O}(n)$:

$$f(x) = 1{,}85889e - 08 + 2{,}68111e - 07 \times x$$

Cuadrático $\mathcal{O}(n^2)$:

$$f(x) = 0{,}00737389 + 2{,}4236e - 07 \times x + 1{,}12009e - 14 \times x^2$$

Logarítmico $\mathcal{O}(n\log(n))$:

$$f(x) = 2{,}4236e - 07 + 1{,}85907e - 08 \times x \times \ln(x)$$

Veamos los ajustes uno por uno, para poder realizar un análisis más minucioso y completo de cada uno de los ajustes.

Lineal (n)

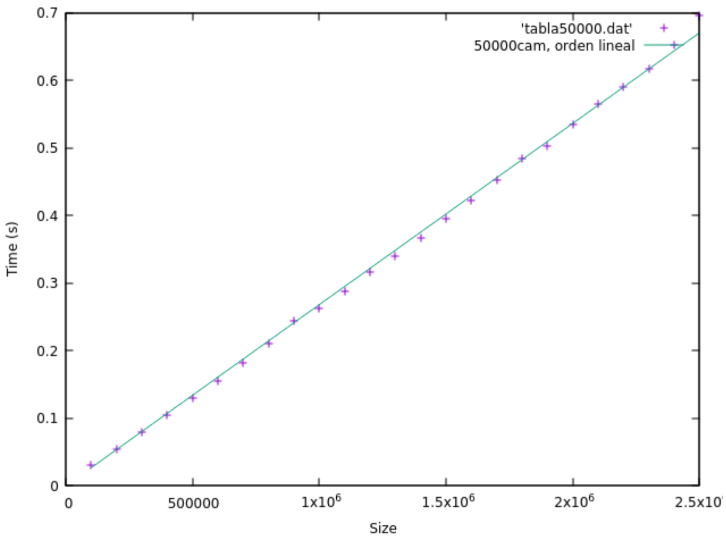

Figura 5: Algoritmo del servicio de catering ejecutado para un total de 50000 camareros, con el ajuste lineal, función del tipo $f(x) = a_1 x + a_0$.

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1               = 2.68111e-07     +/- 2.071e-09     (0.7725%)
a0               = 1.85889e-08     +/- 0.003079      (1.656e+07%)
```

Cuadrático (n^2)

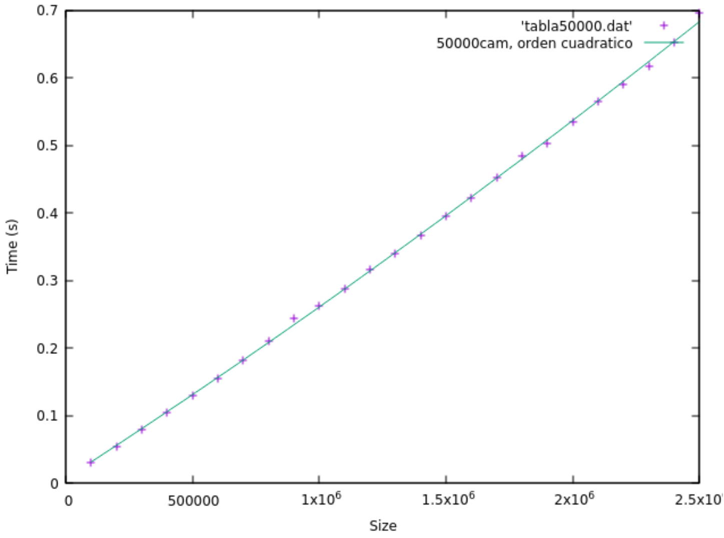

Figura 6: Algoritmo del servicio de catering ejecutado para un total de 50000 camareros, con el ajuste cuadrático, función del tipo $f(x) = a_2x^2 + a_1x + a_0$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 1.12009e-14      +/- 1.871e-15     (16.7%)
a1              = 2.4236e-07       +/- 5.01e-09      (2.067%)
a0              = 0.00737389       +/- 0.002827      (38.34%)
```

Logarítmico ($n\log(n)$)

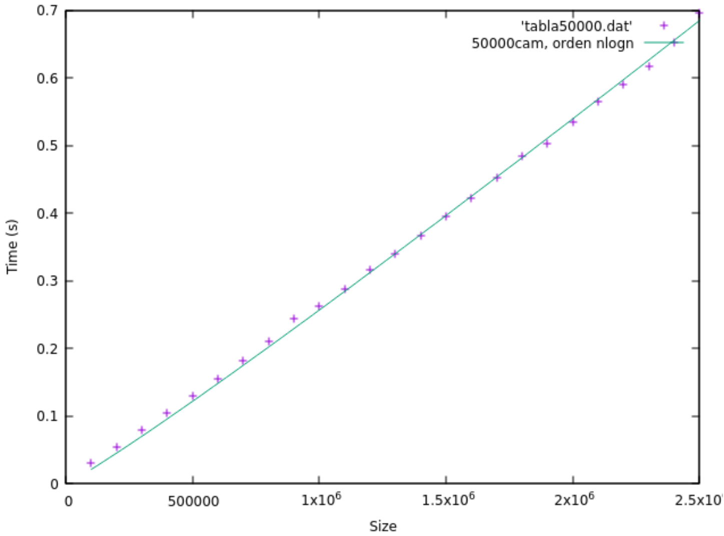

Figura 7: Algoritmo del servicio de catering ejecutado para un total de 50000 camareros, con el ajuste de $n\log(n)$, función del tipo $f(x) = a_1 + a_0 x\log(x)$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 2.4236e-07       +/- 0.002948     (1.216e+06%)
a0              = 1.85907e-08      +/- 1.375e-10    (0.7396%)
```

Con esto quedan recogidos los datos de forma más precisa, pues se les ha aplicado un ajuste con regresión. Hemos obtenido una gran cantidad de información sobre dichas aproximaciones. No obstante, para completar este estudio, usaremos el comando

```
stats 'tabla50000.dat' us 1:2
```

para obtener la siguiente información adicional sobre la bondad de los ajustes.

```
Mean Err.:       144222.0510              0.0392
Std Dev Err.:    101980.3903              0.0277
Skewness Err.:        0.4899              0.4899
Kurtosis Err.:        0.9798              0.9798

Minimum:         100000.0000 [ 0]         0.0316 [ 0]
Maximum:         2.50000e+06 [24]         0.6958 [24]
Quartile:        700000.0000              0.1827
Median:          1.30000e+06              0.3397
```

```
Quartile:      1.90000e+06            0.5025

Linear Model:        y = 2.715e-07 x - 0.005731
Slope:               2.715e-07 +- 1.909e-09
Intercept:           -0.005731 +- 0.002838
Correlation:         r = 0.9994
Sum xy:              1.481e+07
```

5. Problema del Viajante de comercio

5.1. Primera heurística

Pasamos a continuación a comentar la primera heurística que usamos en el problema del viajante de comercio. Dicha heurística se basará en la filosofía "greedy", de forma que obtendremos una solución rápida y relativamente satisfactoria. Dado un vector de localizaciones y la ubicación de la empresa:

1. Creamos el vector solución y añadimos a la empresa. Será nuestro punto de partida.

2. Mientras que el vector no esté vacío, añadimos al vector solución la ubicación más cercana a la última localización del vector solución. Cada vez que añadimos una localización a la solución, la eliminamos del vector de partida. De esta manera se garantiza que el bucle acaba en algún momento.

3. Por último, volvemos a añadir la empresa para obtner un recorrido cerrado. Este paso no tiene mucha importancia y cumple un papel meramente simbólico.

La implementación en **C++** necesita del dato struct *Location*, que representa una ubicación en un plano. Se basará, pues, en dos coordenadas enteras, aunque también incluye constructores y la sobrecarga de ciertos operadores. Por otra parte, resulta imprescindible una función que calcule la distancia entre dos *Location*, así como un método que dado un vector de *Location* y un dato *Location* busque en el vector la ubicación más cercana al dato proporcionado. La función donde se implementa el algoritmo no es más que aplicar esta última función iterativamente hasta vaciar el vector que se suministra.

Comentamos brevemente el paso de datos al programa. El ejecutable recibe un fichero en el que cada línea hay dos enteros. Por simplicidad, asumiremos que la primera línea hace referencia a la ubicación de la empresa. Este archivo se genera de forma automática a partir de otro programa que recibe un númeor entero, que indica el tamaño del programa (número de líneas del fichero que creará). Cabe destacar que los datos se generan de forma aleatoria. Utilícese esta información para las heurísticas venideras.

El código en sí es el que sigue:

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <string>
```

```cpp
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct Location
{
	int x, y;
	Location(int a, int b) : x(a), y(b) {}
	Location() : x(0), y(0) {}
	bool operator==(const Location& p) {
		return (p.x == x) && (p.y == y);
	}
};

// Dadas dos localizaciones, calcula la distancia.
double distance(Location A, Location B){

	double distancia_x = B.x - A.x;
	double distancia_y = B.y - A.y;
	double distancia = sqrt(pow(distancia_x, 2) + pow(
	   distancia_y, 2));

	return distancia;
}

// Dado un vector de Locations y una Location, elimina las
   apariciones de Location del vector.
void eliminate(vector<Location>& v, const Location& l) {
	v.erase(find(v.begin(), v.end(), l));
}

// Dado un vector de Locations, busca la Location más cerca
    a otra Location suministrada.
Location closestLocation (Location A, vector<Location> v){

	Location closest = v[0];
	for (auto i : v) {
		if(distance(i, A) < distance(closest, A)){
			closest = i;
		}
```

```
        }

        return closest;
}

// Primera heurística. Elegimos la localización más cercana de donde nos encontremos.
vector<Location> FirstAprox(vector<Location> customers, Location company){

        vector<Location> path;
        path.push_back(company);

        while(!customers.empty()){
                Location c = closestLocation(path.back(), customers);
                path.push_back(c);
                eliminate(customers, c);
        }
        path.push_back(company);

        return path;
}

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

        // Comprobar que se ha pasado el nombre del fichero como argumento
        if (argc < 2) {
                cout << "Debe proporcionar el nombre del fichero como argumento" << endl;
                return 1;
        }

        // Abrir el fichero
        ifstream file(argv[1]);
        if (!file) {
                cout << "No se pudo abrir el fichero " << argv[1] << endl;
                return 1;
        }

        // Leer la ubicación de la empresa
```

```
        int x, y;
        file >> x >> y;
        Location Company(x, y);

        // Leer las localizaciones de los clientes
        vector<Location> Customers;
        while (file >> x >> y) {
                Location customer(x, y);
                Customers.push_back(customer);
        }

        vector<Location> pru;

        auto start = high_resolution_clock::now(); // Marca de tiempo inicial
        pru = FirstAprox(Customers, Company);
        auto stop = high_resolution_clock::now(); // Marca de tiempo final

        auto transcurrido = duration_cast<duration<double>>(stop - start);
        cout << Customers.size() << "\t" << transcurrido.count() << endl;

        return 0;
}
```

5.1.1. Justificación de su validez (PVC1)

Demostremos la validez de esta primera aproximación. Debemos comprobar que el procedimiento visita cada ciudad exactamente una vez y vuelve al punto de partida. El algoritmo comienza en la empresa y en cada paso selecciona la ubicación más cercana que aún no ha sido visitada, agregándola al recorrido. En cada iteración, se añade una ciudad que aún no visitada hasta ese momento. Al final de las iteraciones, el recorrido que se ha formado visita cada ciudad exactamente una vez y vuelve la localización inicial, es decir, la empresa. Por lo tanto, el algoritmo siempre produce una solución válida.

5.2. Segunda heurística

En esta heurística lo que hacemos es poner en valor los pesos de la arista. Es decir, tenemos en cuenta los valores de las distintas aristas, de las distancias entre las ciudades. En esta, consideramos todas las distancias posibles entre las ciudades (aristas), y escogemos la menor, creando así un ciclo. De esta manera, no importa por cuál se empiece ya que el resultado acaba por ser el mismo.

```cpp
#include <iostream>
#include <cmath>
#include <fstream>
#include <string>
#include <vector>
#include <queue>
#include <algorithm>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct Location
{
	int x, y;

	//Constructores
	Location(int a, int b) : x(a), y(b) {}
	Location() : x(0), y(0) {}

	//Operadores
	bool operator==(const Location& p) const{
		return (p.x == x) && (p.y == y);
	}

	bool operator!=(const Location& p) const {
		return (p.x != x) || (p.y != y);
	}
};

// Dadas dos localizaciones, calcula la distancia.
double distance(Location A, Location B){

	double distancia_x = B.x - A.x;
	double distancia_y = B.y - A.y;
	double distancia = sqrt(pow(distancia_x, 2) + pow(distancia_y, 2));

	return distancia;
```

```
}

struct Edge
{
        Location A, B;
        double distancia;

        //Constructores
        Edge() : A(), B(), distancia(0) {}
        Edge (Location a, Location b) :
        A(a), B(b), distancia(distance(A, B)) {}

        //Operadores
        bool operator==(const Edge& p) const {
                return (p.A == A) && (p.B == B);
        }

        bool operator!=(const Edge& p) const {
                return (p.A != A) || (p.B != B);
        }

};

//Operadores para ordenar. Comprueba si el elemento más a la derecha es menor
// que el elemento más a la izquierda
bool operator<(const Edge& lhs, const Edge& rhs) {
        return lhs.distancia < rhs.distancia;
}

// Función auxiliar que dado un vector de Location y un objeto location, indica
// si ya la tiene incluida.
bool haveIt(vector<Edge> v, Edge l){
        bool found= false;
        int size = v.size();
        Edge e(l.B,l.A);

        for(int i = 0; i < size && !found; i++){
                //Comprobamos si ya contiene dicha arista en alguno de los
                // dos sentidos
                if(v[i] == l || v[i]== e)
```

```
            found = true;
        }

        return found;
}

// Segunda heurística.
vector<Edge> SecondAprox(vector<vector<double>> distances, vector<Location> locations){

        int n = distances.size();
        vector<Edge> edges, path;

        for (int i = 0; i < n; i++) {
                for (int j = i + 1; j < n; j++) {
                        edges.emplace_back(locations[i], locations[j]);
                }
        }
        cout << "Before:" << endl;
        for(auto p : edges){
                cout << "(" << p.A.x << ", " << p.A.y << ") " << endl;
                cout << "(" << p.B.x << ", " << p.B.y << ") " << endl;
        }
        sort(edges.begin(), edges.end());
        cout << "After:" << endl;
        for(auto p : edges){
                cout << "(" << p.A.x << ", " << p.A.y << ") " << endl;
                cout << "(" << p.B.x << ", " << p.B.y << ") " << endl;
        }

        for(const auto e : edges){
                if(path.size() == n-1)
                break;
                if(!haveIt(path, e))
                path.push_back(e);
        }

        return path;
```

```
}

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

        // Comprobar que se ha pasado el nombre del fichero como argumento
        if (argc < 2) {
                cout << "Debe proporcionar el nombre del fichero como argumento" << endl;
                return 1;
        }

        // Abrir el fichero
        ifstream file(argv[1]);
        if (!file) {
                cout << "No se pudo abrir el fichero " << argv[1] << endl;
                return 1;
        }

        // Leer la ubicación de la empresa
        int x, y;
        file >> x >> y;
        Location Company(x, y);

        // Leer las localizaciones de los clientes
        vector<Location> Customers;
        Customers.push_back(Company);
        while (file >> x >> y) {
                Location customer(x, y);
                Customers.push_back(customer);
        }

        //Creamos una matriz de aristas
        vector<vector<double>> grafo;
        vector<double> fila;
        Edge arista;

        for(int i = 0; i < Customers.size(); i++){
                arista.A = Customers[i];
                for(int j = 0; j < Customers.size(); j++){
                        arista.B = Customers[j];
                        arista.distancia = distance(arista.
```

```
                                        A, arista.B);

                                fila.push_back(arista.distancia);
                                //cout << "Fila " << i << " " << fila.size() << endl;
                        }
                        grafo.push_back(fila);
                        fila.clear();
                }
                //cout << "TAM GRAPH: " << grafo.size() << endl;

                vector<Edge> pru;

                auto start = high_resolution_clock::now(); // Marca de tiempo inicial
                pru = SecondAprox(grafo, Customers);
                auto stop = high_resolution_clock::now(); // Marca de tiempo final

                auto transcurrido = duration_cast<duration<double>>(stop - start);
                cout << Customers.size() << "\t" << transcurrido.count() << endl;

                for(auto p : pru){
                        cout << "(" << p.A.x << ", " << p.A.y << ")" << endl;
                        cout << "(" << p.B.x << ", " << p.B.y << ")" << endl;
                }
                //auto duration = duration_cast<seconds>(stop - start); // Cálculo de la duración en segundos
                //cout << "Tiempo de ejecución: " << duration.count() << " segundos." << endl;

                return 0;
}
```

5.2.1. Justificación de su validez (PVC2)

En este caso, debemos evitar que se creen ciclos antes de que hayamos pasado por todas las ciudades y que se pase por una ciudad más de una vez, lo que implica que nunca haya más de dos aristas en cuyos extremos haya una misma ciudad. Por tanto, con este algoritmo tomamos las aristas ordenadas de las más cortas a las más largas verificando que se verifican las condiciones anteriores.

5.3. Tercera heurística

Por último, explicaremos la tercera heurística. La particularidad de esta última heurística reside en que reciclamos el algoritmo para encontrar el polígono convexo de la práctica anterior. Para tener una impresión inicial, veamos el algoritmo como una mezcla entre el algoritmo de Graham y la primera heurística que presentamos. El procedimiento consistirá en, dada una lista de ubicaciones, obtener el camino que recorre los puntos que forman el polígono convexo. Después, sólo quedarán los puntos interiores, y por tanto, no habrá distancias relativamente grandes. Dichos puntos interiores los iremos conectando de la misma manera que la primera heurística.

1. Partimos de una secuencia de ubicaciones. Trivialmente, si tenemos tres puntos o menos, la solución es la propia secuencia. En lo que sigue, el numero de puntos será mayor a tres.

2. Obtenemos un punto que seguro estará en el polígono. Se comprueba de manera inmediata que un punto que tenga una coordenada más grande (en valor absoluto) con respecto a los demás puntos será un vértice del polígono. Nosotros elegiremos el punto que este más abajo (coordenada y mas pequeña). Si hay dos o más puntos a la misma altura, tomaremos el situado más a la izquierda. Llamaremos a este punto p_0.

3. Creamos una lista con todos los puntos excepto p_0.

4. Ordenamos dicha lista con el siguiente criterio: un punto p1 ira delante de otro punto p_2 si el angulo que forma la recta $\overrightarrow{p_1p_0}$ con el eje horizontal es menor que el angulo que forma la recta $\overrightarrow{p_2p_0}$ con el eje horizontal.

5. En esta lista ordenada, colocamos el punto p_0 en la primera posición (los demás elementos se desplazan una posición a la derecha). A esta lista, la llamaremos aux.

6. En una nueva lista (la llamaremos *convex_polygon*), añadimos los tres primeros puntos de la lista aux.

7. Vamos tomando puntos de la lista aux. Cada vez que consideramos un nuevo punto, mientras el punto tomado se sitúe sobre o a la derecha de la recta que delimitan los últimos punto de *convex_polygon*, eliminamos el último punto

de *convex_polygon*. Cuando el punto se sitúe sobre la izquierda de la recta, lo añadimos a *convex_polygon*.

8. Una vez no quedan puntos de aux que considerar, la lista *convex_polygon* será la lista de los puntos que conforman el polígono convexo.

9. Ahora faltan los puntos interiores. A partir del último, punto de *convex_polygon*, empleamos el algoritmo que explicamos en la primera heurística para conectar los puntos restantes.

10. Cuando no queden puntos, tendremos un camino que conecta todas las ubicaciones. Si empezamos desde la empresa, terminaremos en la empresa.

La implementación que llevamos a cabo combina la que realizamos en la primera heurística con el algoritmo específico de la práctica anterior. Lo recordamos brevemente. La ordenación según los ángulos se encarga a la función $sort\_by\_orientation()$. A su vez, empleamos la biblioteca STL con la función $sort()$, y le suministramos una función anónima que recibe dos puntos y devuelve si el resultado de $compare\_by\_orientation()$ es menor que 0. $compare\_by\_orientation()$ es otra función meramente comparativa que indica la orientación relativa entre tres puntos. Se apoya en la función $orientation()$, que dados dos puntos, indica si al considerar otro tercer punto están alineados o si se efectúa un giro en sentido horario o antihorario.

A continuación. mostramos el código:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <fstream>
#include <chrono>
#include <set>

using namespace std;
using namespace std::chrono;


struct Location
{
	int x;
	int y;
	bool operator==(const Location& p) {
		return (p.x == x) && (p.y == y);
```

```
        }
        bool operator<(const Location &other) const // Sobrecargamos el operador < con la semántica siguiente:
        {                                            // un punto p1 es menor que otro punto p2 si p1 está mas abajo,
                if (y != other.y)                     // es decir, tiene componente y menor. En caso de empate,
                {                                     // se escoge el que esté más a la izquierda (componente x menor).
                        return y < other.y;
                }
                else
                {
                        return x < other.x;
                }
        }
        bool operator!=(const Location &other) const // Sobrecargamos el operador !=
        {

                return ((x != other.x) || (y != other.y));
        }
        Location(int a, int b) : x(a), y(b) {}
        Location() : x(0), y(0) {}
};

enum Orientation{
        ALIGNED,
        CLOCKWISE,
        COUNTERCLOCKWISE
};

// Dado un vector de puntos, devuelve el menor punto (ver sobrecarga operador <).
Location FirstLocation(vector<Location> &v)
{
        Location first_p = *min_element(v.begin(), v.end())
```

```
        ;
        return first_p;
}

// Devuelve la orientación relativa entre 3 puntos. Devuelve un valor del tipo de dato enumerado Orientation.
Orientation orientation(const Location p, const Location q, const Location r)
{
        int val = (q.y - p.y) * (r.x - q.x) -                  // Para evitar el cálculo del ángulo, podemos hallar el
        (q.x - p.x) * (r.y - q.y);                             // producto vectorial de los dos vectores que se definen.

        if (val == 0)
        {

                return ALIGNED;                                // Puntos alineados
        }
        else
        {

                return (val > 0) ? CLOCKWISE : COUNTERCLOCKWISE; // Sentido horario o antihorario
        }
}

// Devuelve la distancia entre dos puntos.
double distance(Location a, Location b)
{
        int x_c = b.x - a.x;
        int y_c = b.y - a.y;
        return sqrt(x_c * x_c + y_c * y_c);
}

// Compara dos puntos según el ángulo que forman el eje horizontal y la recta que pasa por
```

```
// cada punto y un punto fijo (p0).
int compare_by_orientation(Location p1, Location p2, Location p0)
{

        Orientation o = orientation(p0, p1, p2); // Hallamos la orientación de los puntos.
        if (o == ALIGNED)
        {
                // Si los puntos están alineados, devuelve el que está más lejos del punto de referencia
                if (distance(p0, p2) >= distance(p0, p1))
                {
                        return -1;
                }
                else
                {
                        return 1;
                }
        }
        else
        {
                // Si los puntos no están en la misma línea, devuelve el que está a la izquierda en relación al punto de referencia
                if (o == COUNTERCLOCKWISE)
                {
                        return -1;
                }
                else
                {
                        return 1;
                }
        }
}

// Ordena un vector según la función anterior.
void sort_by_orientation(vector<Location> &Locations, const Location &P)
{

        sort(Locations.begin(), Locations.end(), [&](const
```

```
            Location &p1, const Location &p2)
        { return compare_by_orientation(p1, p2, P) < 0; });
}

// Calcula el polígono convexo que contine a todos los puntos de un vector.
vector<Location> Convex_Polygon(vector<Location> &Locations)
{

        if (Locations.size() <= 3) {                    // Si hay menos de tres puntos, devolvemos el mismo vector.
                return Locations;
        }

        vector<Location> aux;                           // Vector auxiliar
        Location p0 = FirstLocation(Locations);         // Hallamos el punto más abajo (más a la izquierda en caso de empate)

        for (int i = 0; i < Locations.size(); i++) // Copiamos los demás puntos en el vector auxiliar
        {
                if (Locations[i] != p0)
                {
                        aux.push_back(Locations[i]);
                }
        }

        sort_by_orientation(aux, p0);               // Ordenamos según el ángulo que forme cada punto con el punto de referencia

        aux.insert(aux.begin(), p0);                // Colocamos p0 en la primera posición

        vector<Location> convex_polygon;            // Vector con los puntos que forman el polígono convexo

        convex_polygon.push_back(aux[0]);       // Añadimos
```

```
        los tres primeros puntos del vector auxiliar
        convex_polygon.push_back(aux[1]);
        convex_polygon.push_back(aux[2]);

        for (int i = 3; i < aux.size(); i++)    // Para el resto del vector
        {
                // Si el ángulo que forman los puntos aux[i] y los dos últimos ángulos de convex_polygon
                // hace un giro en sentido horario o los puntos están alineados, eliminamos el último elemento de
                // convex_polygon.

                while (orientation(convex_polygon[convex_polygon.size() - 2], convex_polygon.back(), aux[i]) != 2){

                        convex_polygon.pop_back();
                }
                convex_polygon.push_back(aux[i]);
        }

        return convex_polygon;
}


// Dado un vector de Locations y una Location, elimina las apariciones de Location del vector.
void eliminate(vector<Location>& v, const Location& l) {
        v.erase(remove(v.begin(), v.end(), l), v.end());
}

// Dado dos vectores de Locations, elimina del primero las apariciones de los elementos del segundo.
void eliminateVector(vector<Location>& v, vector<Location>& e) {
        for(int i = 0; i< e.size(); i++){
                v.erase(remove(v.begin(), v.end(), e[i]), v.end());
        }
}
```

```
// Dado un vector de Locations, busca la Location más cerca a otra Location suministrada.
Location closestLocation (Location A, vector<Location> v){

	Location closest = v[0];
	for (auto i : v) {
		if(distance(i, A) < distance(closest, A)){
			closest = i;
		}
	}

	return closest;
}

// Tercera heurística
vector<Location> SecondAprox(vector<Location> &v){

	vector<Location> convex_polygon = Convex_Polygon(v);
	vector<Location> result;
	result = convex_polygon;

	// Eliminamos de v los lementos que ya tenemos procesados
	eliminateVector(v, convex_polygon);

	// Mientras queden elementos, aplicamos la primera heurística
	while(!v.empty()){
		Location c = closestLocation(result.back(), v);
		result.push_back(c);
		eliminate(v, c);
	}

	return result;
}

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

	// Comprobar que se ha pasado el nombre del fichero como argumento
```

```
    if (argc < 2) {
        cout << "Debe proporcionar el nombre del fichero como argumento" << endl;
        return 1;
    }

    // Abrir el fichero
    ifstream file(argv[1]);
    if (!file) {
        cout << "No se pudo abrir el fichero " << argv[1] << endl;
        return 1;
    }

    // Leer la ubicación de la empresa
    int x, y;
    file >> x >> y;
    Location Company(x, y);

    // Leer las localizaciones de los clientes
    vector<Location> Customers;
    while (file >> x >> y) {
        Location customer(x, y);
        Customers.push_back(customer);
    }

    vector<Location> proof;

    auto start = high_resolution_clock::now(); // Marca de tiempo inicial
    proof = SecondAprox(Customers);
    auto stop = high_resolution_clock::now(); // Marca de tiempo final

    auto duration = duration_cast<seconds>(stop - start); // Cálculo de la duración en segundos
    cout << "Tiempo de ejecución: " << duration.count() << " segundos." << endl;

    return 0;
}
```

5.3.1. Justificación de su validez (PVC3)

Veamos que la tercera heurística también es válida. El algoritmo comienza calculando el polígono convexo, sin repeticiones, luego cada ubicación que pertenece al polígono convexo se visita una única vez. Después se aplica la primera heurística, que ya habíamos probado que era válida, luego esta tercera heurística también es correcta. Bien es cierto que se podría argumentar que no se garantiza el comienzo y final en la empresa, pero el algoritmo devuelve un recorrido válido. Lo que habría que hacer sería rotar el vector hasta que la empresa quede en la primera posición del vector, cosa que no hacemos por cuestiones de eficiencia.

***Nota**: nos vemos obligados a hacer referencia al siguiente enlace, en el que se encuentra una fuente de información que nos ha ayudado notablemente para la implementación de estos códigos, facilitando nuestra tarea:*

pinche aquí para ir al enlace en cuestión

5.4. Análisis comparativo de las tres heurísticas

A continuación vamos a realizar una comparación de las tres heurísticas, más concretamente, de los tiempos que hemos obtenido para cada una de ellas al implementarlas para distintos valores de entrada (variando, como siempre, el tamaño de los vectores).

Tenemos los datos que hemos recogido mostrados de diferentes formas. Se pueden consultar en la Tabla 5.4. Al igual que para el problema del Servicio de Catering, los hemos graficado tanto por separado como en una gráfica conjunta. Las gráficas independientes se observan en la Figura 5.4, mientras que la gráfica de la implementación de las tres heurísticas en una sola puede consultarse en la Figura 5.4.

Tamaño	**pvc1**	**pvc2**	**pvc3**
1000	0.047767	0.198842	0.022471
2000	0.191280	0.836079	0.080864
3000	0.423680	1.983360	0.175681
4000	0.752021	3.515040	0.324349
5000	1.185770	5.563160	0.551678
6000	1.715600	8.260440	0.780435
7000	2.329750	11.100500	1.019080
8000	3.032260	14.450700	1.304320
9000	3.831000	18.710300	1.677950
10000	4.828680	23.649700	2.093130
11000	5.834060	29.028000	2.511400
12000	6.818030	35.551700	3.084280
13000	7.931050	41.012700	3.401220
14000	9.094880	48.296500	4.080030
15000	10.582100	54.751900	4.444150
16000	12.369800	59.932700	5.066620
17000	13.979400	70.529000	5.688760
18000	16.459800	77.383700	6.595690
19000	17.593500	87.362700	7.139900
20000	19.589700	96.040000	8.042910
21000	21.935300	105.299000	8.488390
22000	23.787300	116.897000	9.465930
23000	24.707100	125.554000	10.718900
24000	27.195800	143.347000	11.701200
25000	29.482800	155.614000	12.468100

Cuadro 3: Tiempos que ha tardado cada una de las distintas heurísticas que hemos usado para el problema del viajero, en función de los distintos tamaños del vector de entrada.

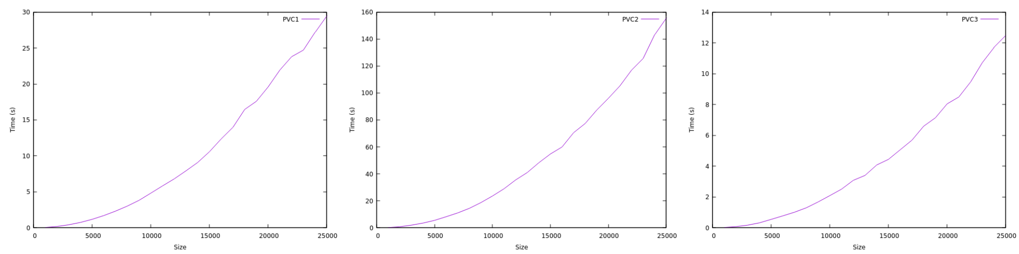

Figura 8: Tiempos obtenidos para las tres heurísticas que hemos implementado de izquierda a derecha: PVC 1, PVC 2, PVC 3.

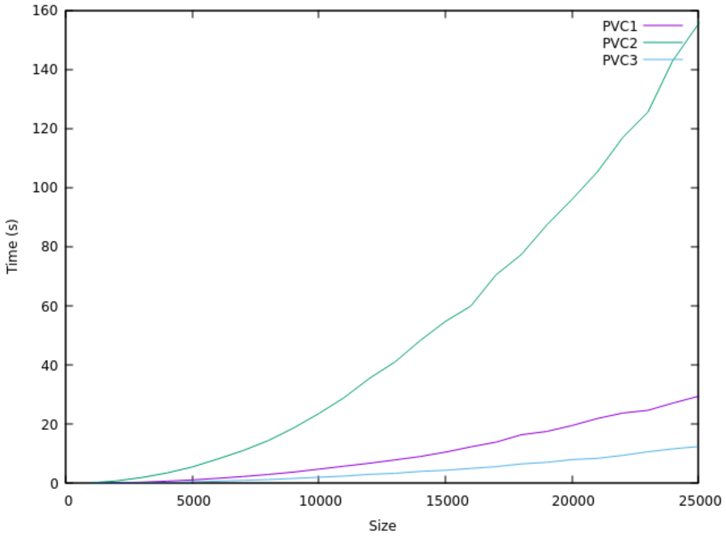

Figura 9: Comparación de las tres heurísticas consideradas.

Sabemos los órdenes de eficiencia de las tres heurísticas: para la primera heurística, se trata de $\mathcal{O}(n^2)$, mientras que para la segunda y tercera, debería ser $\mathcal{O}(n^2 \log(n))$.

En base a los órdenes de eficiencia, en teoría, la heurística menos eficiente debería ser la primera, mientras que las otras dos deben tener una eficiencia similar. Esto lo hemos intentado comprobar posteriormente en el análisis empírico. No obstante, la heurística 2 ha resultado ser la menos eficiente de todas. Probablemente, esto se deba a un error que hemos cometido nosotros en la implementación de las distintas heurísticas.

Por el procedimiento que siguen las mismas heurísticas tenemos que la primera produce resultados dependientes del punto de inicio (de origen), el punto p_0. Toma este como punto departida, y una vez ha pasado por todos los demás, vuelve al mencionado punto p_0.

Por su parte, la segunda produce un resultado independiente del punto de inicio p_0, buscando crear un ciclo a partir de las aristas más cortas.

Por último, en lo referente a la tercera, al igual que la segunda, busca crear un ciclo pero esta vez implementando la envolvente convexa y añadiendo las aristas restantes en los puntos donde suponga un menor costo.

6. Conclusiones

Durante la elaboración de la presente práctica nos hemos enfrentado fundamentalmente a la implementación de un algoritmo “greedy”.

Las principales conclusiones que hemos sacado son las siguientes:

1. Hemos podido comprobar que, realmente, la implementación del algoritmo voraz no es de una gran complejidad. En nuestro caso, lo más relevante ha sido la ordenación del vector, para lo cual hemos usado el *Quicksort*, como ya se vio en su momento.

2. En teoría, la heurística más eficiente debería ser la tercera, lo cual hemos podido comprobar posteriormente en el análisis empírico. No obstante, la heurística 2 es en teoría más eficiente que la 1, aunque luego hemos visto que esto no es así. Probablemente, esto se deba a un error que hemos cometido nosotros en la implementación.

3. También hemos querido resaltar el reciclaje del algoritmo de la práctica anterior, para el cálculo de un polígono convexo, que nos permite idear una nueva heurística, combinada con la filosofía “greedy” de la primera aproximación en el problema del viajante de comercio.

4. El número de camareros que participan en el catering no supone una mejoría para el tiempo de asignación.

5. La resolución del Problema del Viajante de Comercio mediante aproximaciones mejora notablemente el tiempo‘ para la resolución del problema, si bien para tamaños de datos razonablemente grandes los tiempos crecen con una rapidez sorprendente.

En resumen, tras obtener una versión satisfactoria, nos hemos percatado de que la elección de la manera de fraccionar y combinar los sub-problemas (“algoritmos DyV”) resulta clave a la hora de obtener un resultado apropiado‘en caso de que el tamaño de los vectores sea considerablemente grande. Además, de forma implícita hemos podido identificar qué problemas son aptos para ser resueltos a través de esta técnica.