UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 3: Voraces

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

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

27 de septiembre de 2024

Índice

1. Autores

A continuación se va a presentar el trabajo realizado por cada uno de los componentes del grupo.

- Algoritmo voraz: Jaime
- Análisis de eficiencia del algoritmo voraz: Jaime
- Diseño de la primera heurística: Sara
- Diseño de la segunda heurística: Marta
- Diseño de la tercera heurística: Marta
- Análisis comparativo de las tres heurísticas: Sara

Es decir, el trabajo se ha repartido de manera igualitaria, cada uno ha realizado un tercio del trabajo.

2. Objetivos

El objetivo de la práctica es comprender el funcionamiento, implementación e importancia que tienen los algoritmos voraces como método de resolución ante problemas de optimización. En la siguiente práctica se plantean varios problemas que se han de intentar solucionar usando la técnica Greedy y que permitirá mostrar los diferentes alcances de esta técnica.

3. Definición del problema

En esta práctica se van a tratar dos problemas de optimización que se van a intentar resolver usando la técnica Greedy. La filosofía de esta técnica es tomar siempre la mejor decisión en base a criterios locales; es decir, a los datos que se tienen en ese instante. Otra característica importante es que no garantizan alcanzar la solución óptima, para ver esto, en la práctica se van a resolver dos problemas distintos para uno de los cuáles la técnica Greedy no garantiza alcanzar la solución óptima.

3.1. Casos usados para el análisis de eficiencia

Para realizar los análisis de eficiencia empíricos de cada uno de los algoritmos se han medido los tiempos de ejecución para diferentes tamaños del problema.

- **Clientes:** se ha ejecutado el algoritmo desde problemas de tamaño 50000 hasta problemas de tamaño 1250000 con saltos de 50000.
- **Problema del viajante de comercio:** se ha ejecutado el algoritmo desde problemas de tamaño 50 hasta problemas de tamaño 2000 con saltos de 50 en 50, se han tomado tan pocos puntos debido a la efeciencia de los algoritmos.

3.2. Entorno de análisis

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

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

3.3. Descripción del método de medición de tiempos

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

```
high_resolution_clock::time_point t_antes,t_despues;

duration<double> transcurrido;

t_antes = high_resolution_clock::now();

double tiempo_espera_medio= Voraz(camareros, candidatos);

t_despues = high_resolution_clock::now();

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

4. Algoritmo voraz

4.1. Algoritmo diseñado. Descripción del algoritmo

El problema de servicio de catering nos pedía lo siguiente: Una empresa de catering debe servir a 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. Distancia desde la mesa donde se encuentra el comensal a la salida de cocina (desplazamientos).
2. Los comensales que el camarero que le sirve la comida haya tenido que atender con anterioridad.

Objetivo: mediante la técnica Greedy hacer mínimo el tiempo de espera de los clientes.

Elementos del algoritmo Voraz:

1. Conjunto de Candidatos: Clientes, hay que elegir en cada turno que comensal a atender.
2. Conjunto de Seleccionados: los clientes que han sido atendidos.
3. Función Solución: la condición de que todos los clientes hayan sido atendidos.
4. Función de Factibilidad: siempre va a poder alcanzar una solución al problema, es decir atender a todos los clientes.
5. Función Selección: el candidato más prometedor es el cliente más cercano.

6. Función Objetivo: la solución al problema es la media de los tiempos de espera de los clientes.

4.1.1. Diseño del algoritmo

El algoritmo voraz empleado para resolver el problema se basa en lo siguiente:

1. Definimos los clientes como un struct de la siguiente manera:

```
struct Cliente{
    double distancia;
    double tiempo_espera=0;
    //Va a ser (el que le atienda), camarero.distancia_actual
    //+ cliente.distancia
};
```

2. Para los camareros hemos definido un struct también (en este caso no sería tan necesario pero por cuestiones de visibilidad del código lo hemos preferido)

```
    struct Camarero{
        double distancia_actual=0;
    };
```

3. Previamente a la función Voraz, como precondición, lo primero que hacemos es ordenar los clientes de menor a mayor distancia para conseguir la mayor eficiencia (se analiza en la demostración de optimalidad) mediante la siguiente función:

```
    bool orden_dist_cliente (Cliente c1, Cliente c2){
        return(c1.distancia<c2.distancia);
    }
```

Una vez hecho esto recorremos el vector de los clientes y para cada uno vamos calculando su tiempo de espera y actualizando la distancia recorrida por cada camarero. Con respecto a los camareros, vamos asignando uno a cada cliente, y una vez asignados todos, volvemos a coger el primero(sabemos que va a estar libre ya que hemos ordenado los clientes de menor a mayor distancia y vamos asignando los camareros en orden).

Por último, calculamos la media de los tiempos de espera y ese es el valor que devolvemos.

```
double Voraz(vector<Camarero> &camareros, vector<Cliente> &clientes){


    double tiempo_espera_medio;
    double suma_tiempos_espera=0;

    int i=0;
    int j=0;
    while(i < clientes.size()) {

        //Aumentamos el tiempo de espera
        clientes[i].tiempo_espera +=
        camareros[j].distancia_actual + clientes[i].distancia;
        //Actualizamos la suma de todos los tiempos de espera
        suma_tiempos_espera += clientes[i].tiempo_espera;


        //Actualizamos la distancia del camarero j-ésimo
        camareros[j].distancia_actual += 2*(clientes[i].distancia);
        //La ida y la vuelta

        i++;
        j++;

        if (j == camareros.size()) {
            j = 0;
        }
    }

    //Calculamos el tiempo de espera medio
    tiempo_espera_medio = suma_tiempos_espera/clientes.size();

    return(tiempo_espera_medio);
}
```

4.2. Generador de casos del algoritmo voraz

El generador de casos del algoritmo voraz recibe como argumento el numero de clientes, pues se supone el numero de camareros constantes, y para cada cliente se genera de manera aleatoria la distancia desde cocina hasta el lugar en que es atendido cada cliente. Para ello se toma un intervalo de valores y en este caso se ha supuesto que la distancia a la que puede estar un cliente desde la cocina está entre 1 y 100.

```
int main(int argc, char* argv[]) {
    if (argc != 2){
        cerr << "Formato " << argv[0] << " <num_camareros>" << endl;
        return -1;
    }

    int n = atoi(argv[1]);
    //Se toma el número de camareros constante
    int c=10;
    if (n <= c || n>100){
        cerr << "El numero de camareros ha de ser
        mayor estricto que los camareros(10) y menor que 100" << endl;
        return -1;
    }

    ofstream fo;
    fo.open("ficheroDeSalida.txt");
    //Generamos los puntos aleatoriamente
    vector<Cliente> clientes;

    srandom(time(0));
    const int MIN=1, MAX=100;
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<float> distr(MIN,MAX);

    //Le pasamos el numero de camareros
    fo << c << endl;

    int k=0;
    while (k < n) {
        Cliente cliente;
        cliente.distancia = distr(eng);

        fo << cliente.distancia << endl;
        k++;
    }

    fo << endl;
    fo.close();
    return 0;
}
```

4.3. Demostración o justificación de la validez

Como ya hemos comentado, el objetivo del algoritmo es minimizar la media de tiempo de espera de los comensales. Es decir sean:

$$S_1, S_2, ..., S_n$$

los n comensales. Sean c camareros, con $c < n$. Los comensales los hemos ordenado por distancias de menor a mayor, es decir, se cumple que

$$S_1.d <= S_2.d <= ... <= S_n.d$$

No tendría sentido ordenarlos de mayor a menor distancia, cualquier conjunto de valores sirve como contraejemplo. Sea

$$X = (t_{e1}, t_{e2}, ..., t_{en})$$

la solución generada por Voraz para los n comensales. Sean ahora los mismos n comensales pero en un orden cualquiera(que no sea de menor a mayor distancia):

$$S_1', S_2', ..., S_n'$$

A partir de ellos obtenemos una solución factible cualquiera(se ha atendido a todos los comensales)

$$Y = (t_{e1}^{'}, t_{e2}^{'}, ..., t_{en}^{'})$$

La pregunta sería ver si se cumple lo siguiente:

$$\frac{\sum_{i=0}^{n} t_{ei}}{n} <= \frac{\sum_{j=0}^{n} t_{ej}^{'}}{n}$$

Aclaración: Vamos a demostrar que dados unos clientes con sus distancias, el hecho de que haya dos desordenados, es decir, que $S_p > S_q$ con $p < q$ implica que ya va a ser menos óptimo(el tiempo medio de espera va a aumentar) que si estuviesen totalmente ordenado, evidentemente, si hay más de un par de números que estén desordenados, basta repetir el procedimiento y cada vez se irá viendo que es menos eficiente cuando se vaya acercando al caso en el que los clientes están ordenados de mayor a menor distancia.

Vamos a proceder mediante una **Reducción al Absurdo:**
Supongamos que

$$\frac{\sum_{j=0}^{n} t_{ej}^{'}}{n} < \frac{\sum_{i=0}^{n} t_{ei}}{n} \Leftrightarrow \sum_{j=0}^{n} t_{ej}^{'} < \sum_{i=0}^{n} t_{ei} \quad (1)$$

Como en la solución Y no partíamos de que inicialmente los clientes estuviesen ordenados por distancias de menor a mayor $\Rightarrow \exists p \in 1, .., n$ tal que $s_p^{'}.d > s_{p+1}^{'}.d$,(cogemos el primer caso en el que ocurra esto), por simplificar la notación vamos a denotar $p + 1$ como q. De (1) obtenemos

$$\sum_{j=0}^{p-1} t_{ej}^{'} + t_{ep}^{'} + t_{eq}^{'} + \sum_{j=q+1}^{n} t_{ej}^{'} < \sum_{i=0}^{p-1} t_{ei} + t_{ep} + t_{eq} + \sum_{i=q+1}^{n} t_{ei}$$

Pero al ser $s_p^{'}.d > s_q^{'}.d$ para el caso de la solución Y, por como se obtienen los tiempos de espera de los comensales, tenemos que $t_{ep}^{'} > t_{ep}$, lo que implica (*), que para $i > p, t_{ei}^{'} > t_{ei}$ luego llegamos a contradicción (evidentemente ya no se cumple lo que habíamos supuesto, es decir, que la media de los tiempos de espera de la solución Y era menor que los de la solución X) por suponer que la solución Y era más óptima.

-DEM- (*): Si $t_{ep}^{'} > t_{ep} \Rightarrow$ Con $i > p, t_{ei}^{'} > t_{ei}$
Vamos a ver que $t_{eq}^{'} > t_{eq}$, para $i > q$ bastaría una simple inducción.
Sabemos que $t_{eq}^{'} = 2 * s_p^{'}.d + s_q^{'}.d$ y que $t_{eq} = 2 * s_p.d + s_q.d$ (no tenemos en cuenta la distancia recorrida previamente por el camarero ya que es la misma para ambos casos), es ¿$2 * s_p^{'}.d + s_q^{'}.d > 2 * s_p.d + s_q.d$?
Usando las siguientes hipótesis:

1. $s_p = s_q^{'}$
2. $s_q = s_p^{'}$
3. $s_p^{'}.d > s_q^{'}.d$

nos queda:

$2 * s_p^{'}.d + s_q^{'}.d > 2 * s_q^{'}.d + s_p^{'}.d \Leftrightarrow s_p^{'}.d > s_q^{'}.d$. Lo cual es cierto luego queda demostrado.

■

5. Análisis de eficiencia del algoritmo voraz

5.1. Eficiencia teórica

Para estudiar la eficiencia teórica de este algoritmo tenemos que calcular el número de operaciones que se hacen dado un problema de tamaño n. La eficiencia de este algoritmo va a ser el máximo de: la eficiencia de ordenar el vector y la eficiencia de la función Voraz:

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

Con respecto a la función, vemos que todas las sentencias que hay dentro del bucle while son de eficiencia $\mathcal{O}(1)$, pues añadir un elemento a un vector al final y sumar se hace en tiempo constante, supongamos que se ejecuta en un tiempo n. Además tanto la comprobación del if como su interior tienen eficiencia $\mathcal{O}(1)$, luego el interior del bucle tiene eficiencia $\mathcal{O}(1)$. Por tanto, la eficiencia del bucle sería el número de iteraciones que realiza, en este caso n.size() veces, que es el número de clientes en el restaurante; luego realiza n iteraciones.

Por otro lado, el algoritmo sort de la STL es $\mathcal{O}(nlog(n))$ pues utiliza el método de ordenación de quicksort, y como se vio y estudió en la Práctica 1 se trata de un algoritmo con eficiencia $(nlog(n))$. El resto de asignaciones y operaciones son también de orden constante.

Luego nos quedaría que la eficiencia es

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

Contando el número de operaciones de la función se llega a que la función teórica que sigue este algoritmo es de la forma: h(n) = an + b, calculando estas contantes ocultas, mediante un recuento de las operaciones ejecutadas en la función previamente mostrada, tenemos que h(n) = 18n + 8

5.2. Eficiencia empírica

De la misma forma, para estudiar la eficiencia empírica del algoritmo, se ha medido el tiempo de ejecución para distintos tamaños de entrada, en este caso, al tener dos variables(clientes y camareros), hemos dejado fijos los camareros. Los resultados han sido los siguientes:

Tamaño	**Tiempo**
50000	0.00444148
100000	0.00724697
150000	0.0148549
200000	0.0166013
250000	0.019971
300000	0.024379
350000	0.0293625
400000	0.0327202
450000	0.0374312
500000	0.0413933
550000	0.0467736
600000	0.0504957
650000	0.0557602
700000	0.0602508
750000	0.0651408
800000	0.0696349
850000	0.0735876
900000	0.0787168
950000	0.0830667
1000000	0.0875437
1050000	0.0929923
1100000	0.0977357
1150000	0.101167
1200000	0.107534
1250000	0.110425

Cuadro 1: Tabla Voraz para 45000 camareros

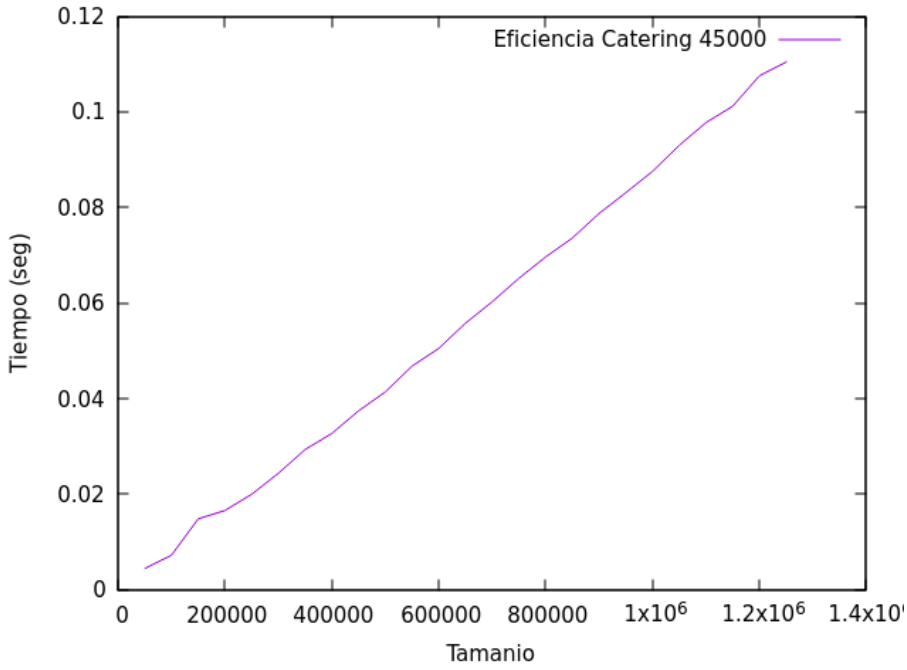

Figura 1: Eficiencia empírica para 45000 camareros

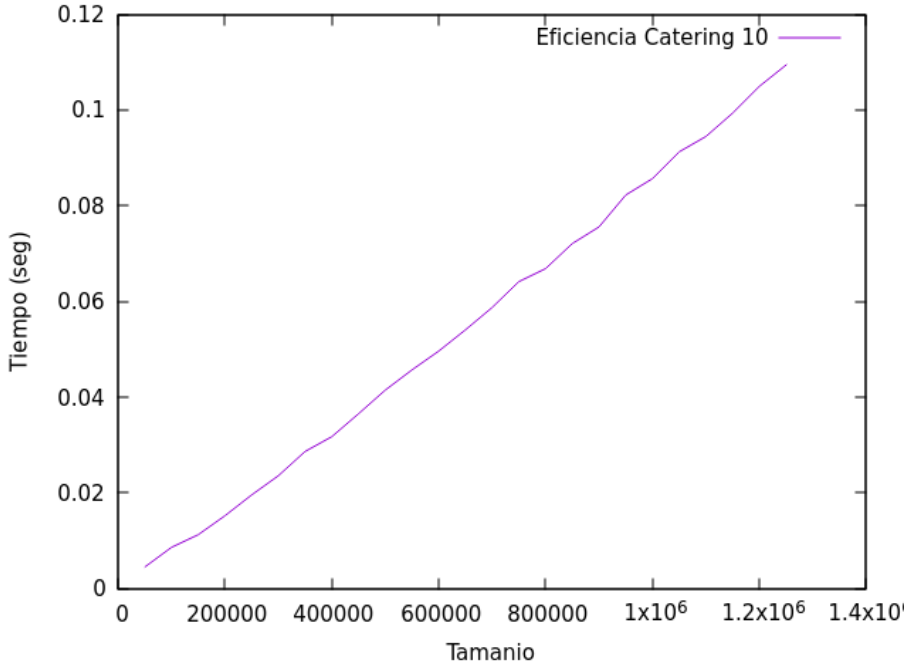

Figura 2: Eficiencia empírica para 10 camareros

Tamaño	**Tiempo**
50000	0.00450528
100000	0.0086513
150000	0.0112698
200000	0.0152395
250000	0.0195894
300000	0.0235981
350000	0.0286784
400000	0.0317571
450000	0.0365406
500000	0.0414949
550000	0.0457176
600000	0.0496334
650000	0.0540685
700000	0.058725
750000	0.06413
800000	0.0668625
850000	0.0721044
900000	0.0755801
950000	0.0822526
1000000	0.085672
1050000	0.0912748
1100000	0.0944841
1150000	0.0993509
1200000	0.104942
1250000	0.109418

Cuadro 2: Tabla Voraz para 10 camareros

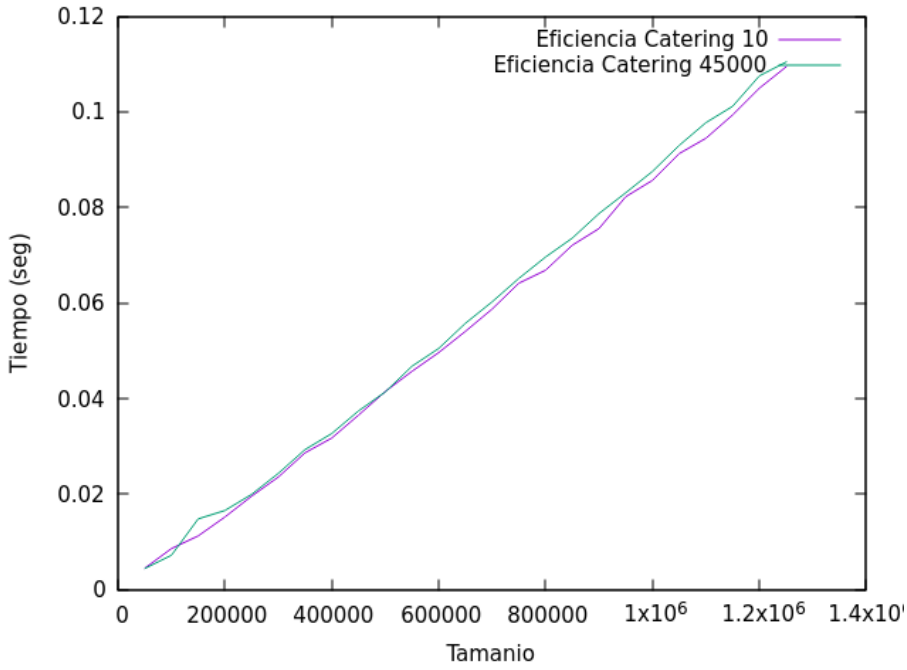

Figura 3: Comparación eficiencia para 10 y 45000 camareros

Conclusión: El uso de un número mayor o menor del número de camareros, no cambia la eficiencia del algoritmo, aunque al usar un número menor de camareros, evidentemente el tiempo medio de espera de los clientes va a aumentar.

5.3. Eficiencia híbrida

La función que ajusta a los datos obtenidos es la siguiente: $f(x) = a_0 \cdot x \cdot log(x)$ Realizando el ajuste por mínimos cuadrados obtenemos:

- Para 10 camareros, la ecuación $f(x) = 6{,}22588 \cdot 10^{-9} \cdot x \cdot log(x)$ con un coeficiente de determinación de $4{,}42749 \cdot 10^{-7}$.
- Para 45000 camareros, la ecuación $f(x) = 6{,}36662 \cdot 10^{-9} \cdot x \cdot log(x)$ con un coeficiente de determinación de $8{,}29528 \cdot 10^{-7}$.

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

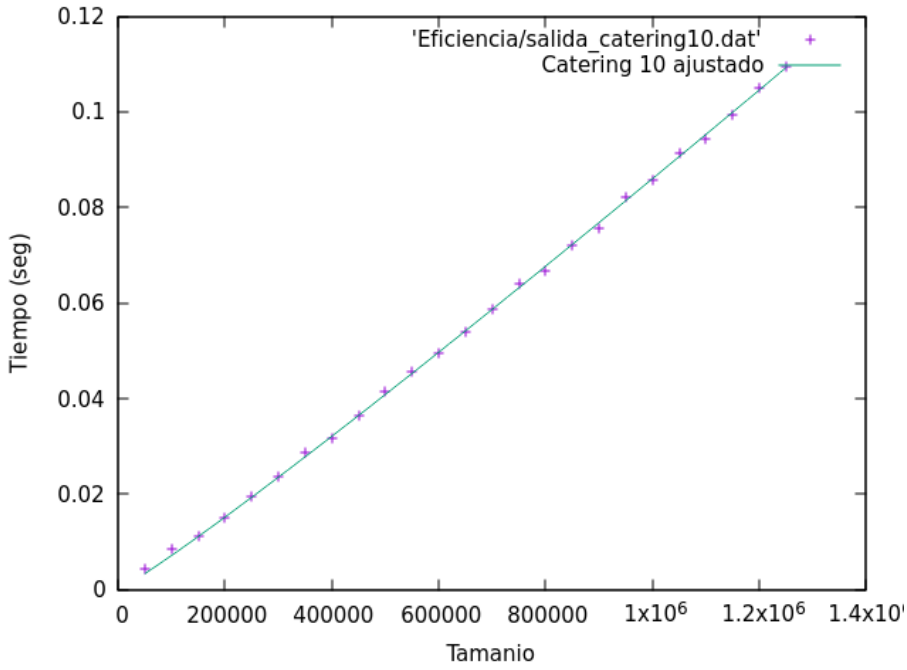

Figura 4: Ajuste gráfica para 10 camareros

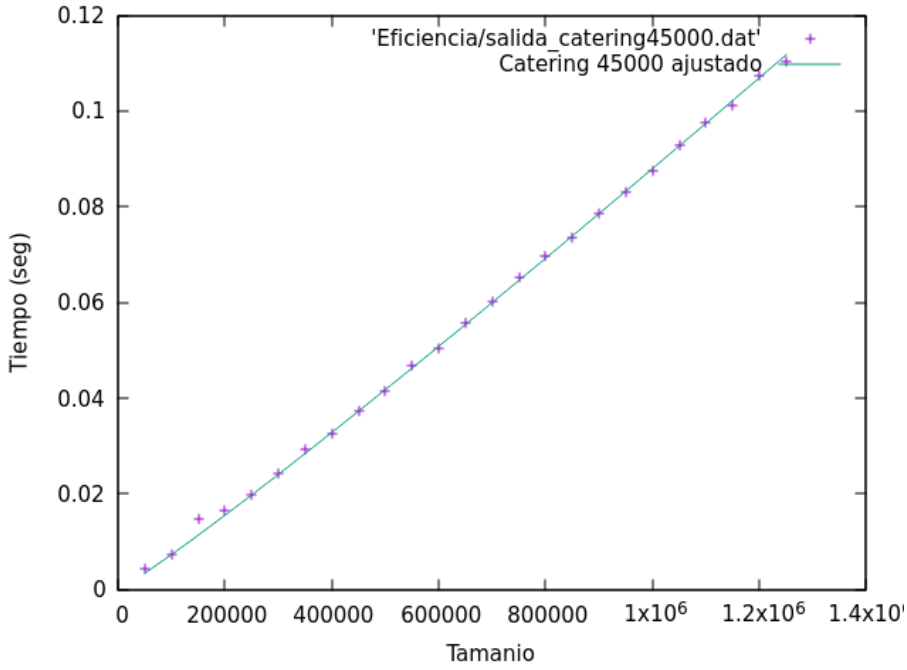

Figura 5: Ajuste gráfica para 45000 camareros

6. Problema del viajante de comercio

El problema del viajante de comercio se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un comerciante tiene que pasar por todas las ciudades una única vez; es decir, sin repeticiones, regresando al punto de partida, lo que supone hacer un ciclo cerrado, minimizando la distancia

recorrida. Una solución a este problema es indicar el orden en que deben recorrerse las ciudades y sumar las distancias recorridas, incluyendo aquella que cierra el ciclo. Se trata de un problema NP. Mediante la técnica Greedy no se garantiza alcanzar la solución óptima, pero si alcanzar una solución mediante algoritmos aproximados. Es por ello que a continuación se exponen tres heurísticas que nos llevan a la solución, la cual no siempre será la óptima, y que posteriormente compararemos en términos de eficiencia y de calidad de las soluciones.

7. Generador de casos del problema del viajante de comercio

Para poder probar cada una de las heurísticas en primer lugar hay que generar las distancias entre todas las ciudades. Por definición del problema vamos a suponer que estamos antes un grafo completo; es decir todas las ciudades están conectadas entre sí. Por otro lado como dato de entrada el generador de casos lee el número de ciudades. A partir de ahí genera las distancias aleatorias que se van almacenando en una matriz simétrica, teniendo en cuenta que la diagonal principal de la matriz está rellena de ceros.
Los datos generados son almacenados en un fichero desde el cual cada heurística los leerá para ejecutar su código.

A continuación, se presenta el código que se ha utilizado para la generación de los datos.

```
int main(int argc, char* argv[]) {
    if (argc != 2){
        cerr << "Formato " << argv[0] << " <num_ciudades>" << endl;
        return -1;
    }
    int n = atoi(argv[1]);
    if (n < 0){
        cerr << "El numero de ciudades ha de ser positivo" << endl;
        return -1;
    }
    ofstream fo;
    fo.open("ficheroDeSalida.txt");
    // Mandar el numero de ciudades
    fo << n << endl;
    //Generamos los puntos aleatoriamente
    srandom(time(0));
    const int MIN = 1, MAX = 1000;
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<float> distr(MIN, MAX);

    double distancia;
    vector<vector<double>> matriz;
    vector<double> aux;

    for(int i=0; i<n; i++){
        aux.push_back(0.0);
    }
    for(int i=0; i<n; i++){
        matriz.push_back(aux);
    }

    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(i<j){
                distancia = distr(eng);
                matriz.at(i).at(j) = distancia;
                matriz.at(j).at(i) = distancia;
            }
            fo << setw(10 ) << matriz.at(i).at(j) << " ";
        }
        fo << endl;
    }
    fo << endl;
    fo.close();
    return 0;
}
```

8. Diseño de la primera heurística para el PVC

La primera heurística elegida para la resolución del problema del viajante de comercio basa su diseño en algoritmo de Kruskal explicado en clase que tiene como objetivo pasar por todos los puntos de un grafo sin crear ciclos y con el camino mínimo eligiendo las aristas más cortas. Sin embargo, en el problema del viajante de comercio el objetivo es pasar por todas las ciudades una única vez y que el ciclo final sea cerrado. Para ello dada la matriz simétrica con las distancias entre las ciudades se selecciona tan solo la triangular superior y se almacena en un vector de list, pues es más eficiente ante inserción y borrados.

La función recibe una matriz de aristas, teniendo en cuenta que una arista es un struct que guarda los puntos que une y su longitud, es posible ordenar las listas sin perder que puntos alcanzan éstas. Una vez ordenadas, la idea es ir eligiendo las aristas con distancia menor y esta arista podrá ser añadida al vector solución siempre y cuando los puntos que se unan se encuentren en componentes conexas distintas (para evitar ciclos) y que al añadir la arista de un punto no salgan más de dos aristas (para asegurarnos de que tan solo se pasa una vez por cada ciudad). Al inicio del algoritmo se define un conjunto de componentes formadas cada una por un único punto y cada vez que se añade una arista se deben fusionar las componentes conexas de los vértices que une dicha aristas y eliminar los dos conjuntos anteriores.

Para esta heurística a pesar de que el dato recibido es una matriz con las distancias como se trata de ir eligiendo las mejores aristas, para resolverlo de forma más eficiente se ha creado el siguiente struct y se han almacenado los datos en un vector de list, guardando tan sólo los datos de la triangular superior.

```
struct Vertice{
    int id;
    bool operator == (Vertice p) const{
        return (id == p.id);
    }
    bool operator != (Vertice p) const{
        return (id != p.id);
    }
    bool operator < (Vertice p) const{
        return (id < p.id);
    }
};
struct Arista{
    double distancia;
    Vertice punto1;
    Vertice punto2;
    bool operator<(Arista p) const{
        return (distancia < p.distancia);
    }
};
```

En primer lugar, se van a mostrar unas funciones que son utilizadas para el algoritmo y que también serán explicadas.
Función FindSet: se encarga de encontrar la componente conexa a la que pertenece un vértice, en este caso una ciudad.

```
set<Vertice> findSet( set<set<Vertice>>  & componentes,
                                    const Vertice & p){
    bool encontrado= false;
    set<set<Vertice>>::iterator it = componentes.begin();
    set<Vertice> resultado;
    while(it != componentes.end() && !encontrado){
        set<Vertice>::iterator iter;
        iter = (*it).find(p);
        if(iter != (*it).end()){
            encontrado = true;
            resultado = *it;
        }
        else ++it;
    }
    return resultado;
}
```

Funcion Union: fusiona dos componentes conexas; es decir, dos conjuntos en uno único sin repeticiones.

```
set<Vertice> Union(set<Vertice> setU, set<Vertice> setV){
    set<Vertice>:: iterator it;
    for(it = setV.begin(); it!=setV.end(); ++it)setU.insert(*it);

    return setU;

}
```

Función setDifferent: evuelve true si dos componentes conexas conexas son distintas; es decir, si no tienen ningun punto en común.

```
bool setDifferent(set<Vertice> setA, set<Vertice> setB){

    set<Vertice>::iterator it = setA.begin();
    set<Vertice>::iterator iter;
    bool son_diferentes =true;

    while(it != setA.end() && son_diferentes){
        iter = setB.find(*it);
        if(iter == setB.end())  ++it;
        else son_diferentes = false;
    }

    return son_diferentes;
}
```

Funcion UnVerticeDosAristas: comprueba que desde un vértice no partan más de dos aristas para que el camino pueda ser cerrado.

```
bool UnVerticeDosAristas(vector<int> vertices, Arista arista){
    bool un_vertice_tiene_dos_aristas = false;

    if(vertices.at(arista.punto1.id -1) ==2 ||
                vertices.at(arista.punto2.id -1) ==2 )
               un_vertice_tiene_dos_aristas=true;
    return un_vertice_tiene_dos_aristas;
}
```

A continuación se muestra el código del algoritmo

```
vector<Arista> PVC1(const vector<Vertice> & vertices,
                                vector<list<Arista>> & aristas){
    vector<list<Arista>> aristas_ordenadas = aristas;
    //Crear un set con cada vertice, cada componente conexa
    int n = vertices.size();
    set<set<Vertice>> componentes;

    for(int i = 0; i < vertices.size(); i++){
        set<Vertice> conjunto;
        conjunto.insert(vertices.at(i));
        componentes.insert(conjunto);
    }
    // Ordenar cada vector de aristas

    vector<list<Arista>>::iterator it;
    for(it=aristas_ordenadas.begin();
                    it!=aristas_ordenadas.end();++it){
        (*it).sort();
    }

    vector<Arista> conjunto_salida;
    vector<int> vertices_aristas_salientes;
    for(int i = 0; i < aristas_ordenadas.at(0).size()+1; i++)
            vertices_aristas_salientes.push_back(0);

    while(conjunto_salida.size()<n-1 &&!aristas_ordenadas.empty()){

        //Sacar la arista minima
        Arista minima = aristas_ordenadas.at(0).front();

        int i = 0;
        int num =0;
        vector<list<Arista>>::iterator it=aristas_ordenadas.begin();
        vector<list<Arista>>::iterator fila = it;
        for(it = aristas_ordenadas.begin();
                it != aristas_ordenadas.end(); ++it){

            if((*it).front() < minima ) {
                num = i;
                fila = it;
                minima = (*it).front();
            }
            i++;
        }

        (*fila).pop_front();
        if(aristas_ordenadas.at(num).empty()){
            aristas_ordenadas.erase(fila);
        }
```

```
        // Buscar en que componentes estan los vertices
        // de la arista seleccionada

        Vertice u = minima.punto1;
        Vertice v = minima.punto2;
        set<Vertice> setU = findSet(componentes, u);
        set<Vertice> setV = findSet(componentes, v);

        if(setDifferent(setU,setV)) {
            if (!UnVerticeDosAristas(vertices_aristas_salientes, minima)) {
                vertices_aristas_salientes.at(minima.punto1.id - 1)++;
                vertices_aristas_salientes.at(minima.punto2.id - 1)++;
                conjunto_salida.push_back(minima);
                componentes.erase(setU);
                componentes.erase(setV);
                set<Vertice> nuevo = Union(setU, setV);
                componentes.insert(nuevo);
            }
        }
    }

    Vertice u;
    Vertice v;

    int id1 ;
    bool encontrado = false;
    int i = 0;
    while (i < vertices_aristas_salientes.size() && !encontrado){
        if(vertices_aristas_salientes.at(i) == 1){
            id1 = i+1;
            encontrado = true;
            vertices_aristas_salientes.at(i)++;
        }
        else i++;
    }
    u.id = id1;

    int id2 ;
    encontrado = false;
    i = 0;
    while (i < vertices_aristas_salientes.size() && !encontrado){
        if(vertices_aristas_salientes.at(i) == 1){
            id2 = i+1;
            encontrado = true;
            vertices_aristas_salientes.at(i)++;
        }
        else i++;
    }
    v.id = id2;
```

```
    Arista ultima;

    if(u.id < v.id){
        ultima.punto1 = u;
        ultima.punto2 = v;
        list<Arista>::iterator it= aristas.at(u.id -1).begin();
        bool encontrado = false;
        while( it != aristas.at(u.id -1).end() && !encontrado){
            if((*it).punto2.id == v.id){
                ultima.distancia = (*it).distancia;
                encontrado = true;
            }
            else ++it;
        }
     }
    else{
        ultima.punto1 = v;
        ultima.punto2 = u;
        list<Arista>::iterator it= aristas.at(v.id -1).begin();
        bool encontrado = false;
        while( it != aristas.at(v.id -1).end() && !encontrado){
            if((*it).punto2.id == u.id){
                ultima.distancia = (*it).distancia;
                encontrado = true;
            }
            else ++it;

        }
    };

    conjunto_salida.push_back(ultima);

    return conjunto_salida;

}
```

Al tratarse de un algoritmo voraz se van a describir a continuación las características propias de él presenten en el algoritmo:

- **Conjunto de Candidatos:** vector de listas que recibe la función, denominado *aristas*
- **Conjunto de seleccionados:** vector de aristas que se devuelve, llamado **conjunto salida**
- **Función solución:** comprobación del while, pues se habrá alcanzado la solución cuando el tamaño del conjunto de salida sea iguala que el número de

vértices. En concreto, una unidad inferior porque la última arista, la que cierra el ciclo se mete al final.

- **Función de factibilidad:** se trata de las dos condiciones que se evalúan en el while. por una lado, si las componentes conexas de los vértices que forman la arista que se quiere unir son diferentes, pues en casos de no serlo se provocaría un ciclo. Y por otro lado, la comprobación de que desde un vértice no pueden salir más de dos aristas pues en caso contrario se estaría pasando mas de una vez por una misma ciudad, lo que cual va en contra del enunciado del problema.
- **Función de selección:** tomar la arista más pequeña del conjunto de candidatos
- **Función objetivo:** da el valor de la solución alcanzada, devuelve un vector con las aristas que conforman el camino a seguir.

Para mostrar la solución de forma correcta se llama a la función Mostrar Camino, que muestra el orden en que se han de recorrer las ciudades con el objeto de encontrar una camino cerrado. Devuelve un vector con los identificadores de las ciudades en la posición en que se han de recorrer.

```
vector<int> MostrarCamino( vector<Arista> &aristas ){
    vector<int> camino_final;
    Arista a =aristas.front();

    camino_final.push_back(a.punto1.id);
    camino_final.push_back(a.punto2.id);

    Vertice v = a.punto2;
    aristas.erase(aristas.begin());

    while(!aristas.empty()){
        bool encontrado = false;
        vector<Arista>::iterator it;
        for(it=aristas.begin();it!=aristas.end()&&!encontrado;++it)
            if((*it).punto1==v || (*it).punto2==v){
                encontrado = true;
                if((*it).punto1==v ){
                    camino_final.push_back((*it).punto2.id);
                    v= (*it).punto2;
                }
                else {
                    camino_final.push_back((*it).punto1.id);
                    v = (*it).punto1;
                }
                aristas.erase(it);
            }
        }
    }
    return camino_final;
}
```

8.1. Justificación de la validez

Como ya se expresó en la definición del ejercicio, el problema del viajante de comercio se trata de un problema que no puede encontrar una solución óptima mediante el algoritmo Greedy; es decir, aplicando esta heurística no se garantiza llegar a la mejor solución. Si bien es cierto que al tratarse de una heurística se pretende encontrar soluciones buenas a problemas concretos sin que sean óptimas. Teniendo en cuenta como se hace la elección de las aristas se puede garantizar que bajo las condiciones del problema, que las n-1 aristas seleccionadas son las mejores que se pueden tomar. La acotación sobre lo bueno o malo que es el problema se tiene en la elección de la ultima arista, pues puede ser la peor de todas.

8.2. Análisis de eficiencia

8.2.1. Análisis teórico

El cálculo de la eficiencia teórica se basa en contabilizar el número de operaciones que se van a llevar a cabo en función del tamaño del problema. En este caso tratándose de un problema donde los el tamaño depende tanto del número de aristas como de vértices, denotaremos por V al número de vértices y por A al número de aristas. Es importante conocer que el número de aristas al ser un grafo completo es de V(V-1)/2. Primero debemos conocer la eficiencia de las funciones que son llamadas en el algoritmo:

1. **Función Union:** recorre un set que máximo tendrá el tamaño del numero de vértices del problema, luego $\mathcal{O}(V)$
2. **Función findSet:** recorre todos los set hasta encontrar aquel al que pertenece el vértice, en el peor de los casos dará tantas iteraciones como vértices, luego $\mathcal{O}(V)$
3. **Función setDifferent:** recorre a lo sumo un set de tamaño como máximo n: el tamaño del problema, luego $\mathcal{O}(V)$
4. **UnVerticeDosAristas:** realiza dos comprobaciones, luego $\mathcal{O}(1)$

Al principio del algoritmo se ordena un vector formado por V listas cada una a lo máximo de V componentes, teniendo en cuenta que ordenar listas, se hace con una eficiencia $\mathcal{O}(VlogV)$, ordenar V listas de tamaño V, se hace en tiempo $\mathcal{O}(V^2log(V))$. Dentro del while todas las acciones que se ejecutan se hacen en un tiempo lineal, respecto del número de vértices $\mathcal{O}(V)$. El while itera sobre el número de aristas, luego la eficiencia de este es $\mathcal{O}(V^2 * A) = \mathcal{O}(V)$. Y las órdenes para encontrar la última se hacen en tiempo lineal, con respecto al número de aristas o vértices. Por todo esto, la eficiencia teórica es $\mathcal{O}(V^3)$.

8.2.2. Análisis empírico

Para estudiar la eficiencia empírica se va a mostrar una tabla con los resultados de tiempo de ejecución, teniendo en cuenta que se trata de un algoritmo con eficiencia teórica$\mathcal{O}(V^3)$ se ha hecho un estudio del tiempo para un tamaño reducido de casos. Se ha realizado iteraciones entre 10-2000 con saltos de 50 en 50.

Tamaño	Tiempo
50	0.00244559
100	0.00470864
150	0.0212251
200	0.0648226
250	0.0463576
300	0.156311
350	0.645985
400	0.543095
450	0.624432
500	0.923133
550	1.07908
600	1.09428
650	2.74742
700	3.61451
750	7.7536
800	10.0575
850	5.61867
900	12.9607
950	9.28306
1000	20.5682
1050	15.6475
1100	29.2588
1150	59.4058
1200	6.6205
1250	48.1589
1300	40.7835
1350	78.6503
1400	105.524
1450	93.5514
1500	91.904
1550	119.092
1600	124.774
1650	138.633
1700	214.58
1750	153.198
1800	141.605
1850	68.7349
1900	370.283
1950	342.995
2000	454.251

Cuadro 3: Tabla tiempos PVC1

8.2.3. Análisis híbrido

Para la eficiencia híbrida se calculará la función que mejor ajuste a los puntos. Teniendo en cuenta la eficiencia teórica la ecuación obtenida ha sido la siguiente: f(x) $= t(n) = 1{,}07494 \cdot 10^{-7} \cdot n^3 - 0{,}00015912 \cdot n^2 + 0{,}0743952 \cdot n - 8{,}01546$ y la gráfica con el ajuste es la siguiente.

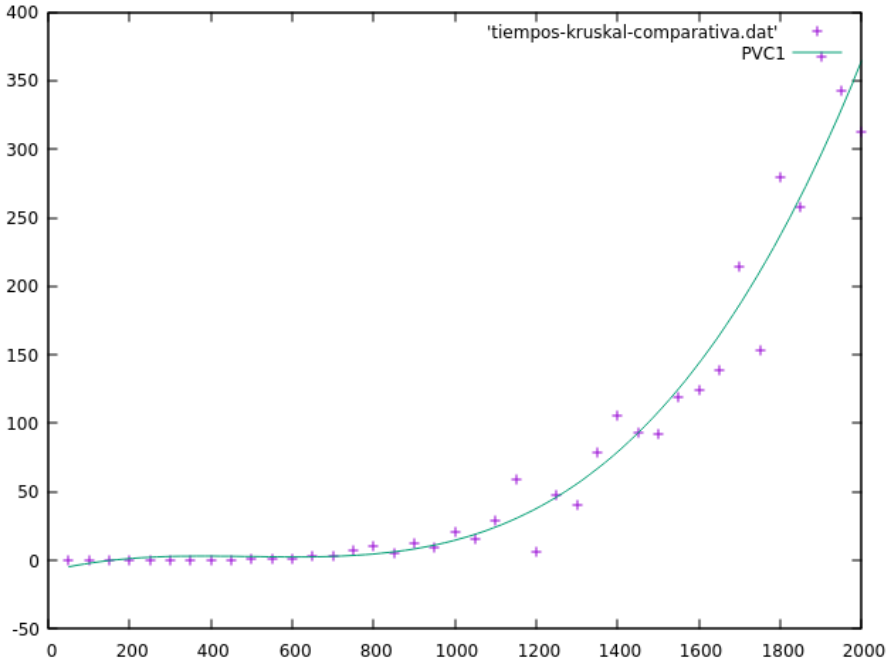

Figura 6: Gráfico función eficiencia híbrida

9. Diseño de la segunda heurística para el PVC

La segunda heurística toma otro enfoque para tratar de resolver el Problema del Viajante de Comercio, puesto que ahora los nodos son los candidatos.

En primer lugar, hay que destacar que en este caso se vuelve a disponer de un número fijo de vértices, n, cuyas distancias entre sí se continúan almacenando en una matriz simétrica. De esta forma si tenemos la matriz $A = (a_{ij})$, con $i \in \{1, ..., n\}$ y $j \in \{1, ..., n\}$, el elemento $a_{ij} = a_{ji}$ guarda la distancia entre los vértices i y j que siempre es un valor decimal estrictamente mayor que 0 pues $a_{ij} = 0$ sí y solo sí $j = i$.

Para simplificar y mejorar la eficiencia del código, dicha matriz ha sido almacenada en un vector de vectores de Aristas la biblioteca STL de C++. El tipo de dato Arista es a su vez un struct que contiene los dos vértices que une y la distancia entre ellos.

```
struct Arista{
    double distancia;
    int p1;
    int p2;
};

vector<vector<Arista>> matriz_dist;
```

La idea del algoritmo se basa en el de Prim pero adaptándolo a nuestro tipo de problema. Para implementarlo se han usado varias funciones.

- La función **Suma_Distancias** calcula el valor total de todas las aristas almacenadas en un vector del tipo Arista, es decir la distancia total recorrida por el Viajante de Comercio.

```
double Suma_Distancias(const vector<Arista> &v){

    double suma = 0.0;

    for(int i=0; i<v.size();i++){
        suma += v.at(i).distancia;
    }

    return suma;
}
```

- La función **Busca_Camino** es la principal. Consiste en tomar el primer vértice guardado y en cada iteración avanzar al vértice más próximo que no haya sido visitado. Cada vez que se cambia a un nodo diferente se guarda la arista elegida en un vector del tipo Arista que al final se devuelve como solución al problema. De esta forma se consigue recorrer todos los nodos ciudades sin repetir ninguno y sin crear ciclos. Es importante que al final del algoritmo se una el nodo final con el nodo de inicio, pues el viajante debe regresar al punto de partida.

```
vector<Arista> Busca_Camino(const vector<vector<Arista>> &matriz){

vector<Arista> aristas;
vector<bool> vertices_seleccionados;
int num_vertices = matriz.size();

for(int i = 0; i< num_vertices; i++)
    vertices_seleccionados.push_back(false);

vertices_seleccionados.at(0) = true;
int pos_actual = 0;

while(aristas.size()<num_vertices){

    // Busco la arista mínima conectada a ese vértice
    double min = 9999999;
    int pos_min = pos_actual;

    for (int i = 0; i < num_vertices; ++i) {
        double d = matriz.at(pos_actual).at(i).distancia;
        if(d<min && d>0 && !vertices_seleccionados.at(i)){
            min = d;
            pos_min = i;
        }
    }

    vertices_seleccionados.at(pos_min) = true;
    aristas.push_back(matriz.at(pos_actual).at(pos_min));
    pos_actual = pos_min;
}

aristas.back()= matriz.at(pos_actual).at(0);

return aristas;

}
```

9.1. Justificación de la validez

Esta heurística en la que se aplica la filosofía Greedy sigue sin ser la solución óptima del problema pues no es posible hallarla, pero parece una buena opción para encontrar una solución factible. El problema puede residir en la elección del punto de partida, pues siempre se toma es el primer nodo guardado que puede no derivar al camino más corto. Sin embargo; el algoritmo es válido pues siempre se recorren todos los vértices, volviendo siempre al origen, y se toma la mejor opción en cada

momento.

Se especifican a continuación donde se presentan en el algoritmos los elementos propios de la técnica Greedy:

- **Conjunto de Candidatos:** para cada vértice un vector del tipo Arista donde se almacenan las distancias a otros vértices (es cada fila del parámetro **matriz** que se pasa a la función Busca_Camino.
- **Conjunto de seleccionados:** vector del tipo Arista que se devuelve, llamado **aristas**.
- **Función solución:** comprobación del while, pues se habrá alcanzado la solución cuando el tamaño del conjunto de salida sea igual que el número de vértices. En concreto, una unidad inferior porque la última arista, ya que el cierre del ciclo se realiza al final.
- **Función de factibilidad:** se trata de las condiciones para encontrar una nueva arista mínima para poder añadirla al conjunto de soluciones. Estas condiciones son que la arista tenga distancia sea estrictamente positiva y que el vértice no haya sido seleccionado.
- **Función de selección:** tomar la arista más pequeña del conjunto de candidatos
- **Función objetivo:** la función Crea_Camino que se especifica más tarde da el valor de la solución alcanzada, devuelve un vector con las aristas que conforman el camino a seguir.

9.2. Análisis de eficiencia

9.2.1. Análisis teórico

Para analizar la eficiencia teórica de la heurística, estudiemos antes la eficiencia de las funciones que intervienen, en función del número de vértices introducidos al que denotamos por n:

1. Función Busca_Camino: su eficiencia es $\mathcal{O}(n^2)$ pues tomamos el máximo entre el primer bucle con eficiencia $\mathcal{O}(n)$ pues se realiza n veces una inserción que es $\mathcal{O}(1)$ y el segundo bucle que contiene otro bucle anidado donde ambos llegan hasta n iteraciones luego en el peor caso su eficiencia es $\mathcal{O}(n^2)$.

2. Función Suma_Distancias: su eficiencia es $\mathcal{O}(n)$ ya que recorre el vector completo de aristas cuyo tamaño como máximo es n, el número de vértices y en cada realiza una suma cuya eficiencia es constante. En consecuencia, se deduce que la eficiencia teórica es

$$T(n) = \mathcal{O}(n) + \mathcal{O}(n^2) = max(\mathcal{O}(n), \mathcal{O}(n^2)) \in \mathcal{O}(n^2)$$

9.2.2. Análisis empírico

Num. vértices	Tiempo (seg)
100	5.4329e-05
150	0.000119365
200	0.000340979
250	0.000404364
300	0.000643492
350	0.000640686
400	0.000824725
450	0.00119209
500	0.00126061
550	0.0014951
600	0.00193812
650	0.00175409
700	0.0014824
750	0.00110681
800	0.00226617
850	0.00170115
900	0.00196876
950	0.00232592
1000	0.00302991
1050	0.0021274
1100	0.00276316
1150	0.00317014
1200	0.00333406
1250	0.0036557
1300	0.00393714
1350	0.00397927
1400	0.00373098
1450	0.00390877
1500	0.00484368
1550	0.00516912
1600	0.00474714
1650	0.00567809
1700	0.00582478
1750	0.00624353
1800	0.00682874
1850	0.00626675
1900	0.0071769
1950	0.0097108
2000	0.00797142

9.2.3. Análisis híbrido

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

n. Al realizar el ajuste se obtienen para 2000 entradas en saltos de 50, la ecuación: $t(n) = 1{,}73533 \cdot 10^{-9} \cdot n^2 + 4{,}15215 \cdot 10^{-7} \cdot n + 0{,}000314032$ con un coeficiente de determinación de $2{,}316 \cdot 10^{-7}$.
Como el residuo es casi cero vemos que el ajuste es muy bueno.

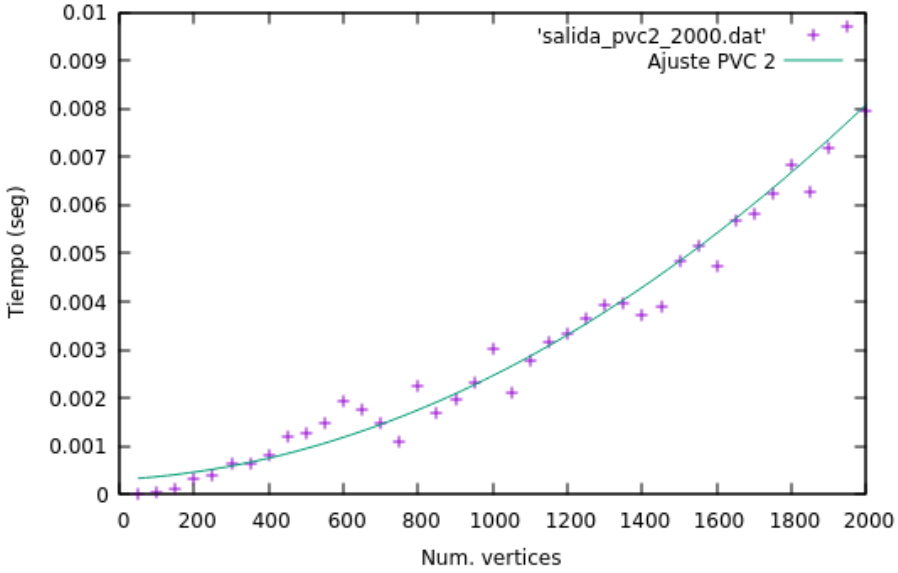

Figura 7: Gráfica del análisis empírico del algoritmo PVC2 para 2000 vértices en saltos de 50

10. Diseño de la tercera heurística para el PVC

La tercera heurística surge de la anterior, donde siempre tomábamos como nodo inicial el primero guardado; sin embargo, no siempre obteníamos el mejor resultado posible pues cabía la posibilidad de que existiese otro recorrido más corto. Por tanto, esta es la idea fundamental de este tercer algoritmo: buscar la unión de aristas que sume menor distancia tomando como punto de partida cada uno de los vértices posibles y elegir la mejor opción.

Del otro algoritmo se mantienen los datos de entrada que continúan siendo el número de vértices y las distancias entre estos almacenadas en una matriz simétrica y el tipo de dato Arista guardado en un struct. Además, se vuelve a utilizar la función **Suma_Distancias** especificada anteriormente.

No obstante, se añaden dos funciones nuevas que llevan a cabo lo explicado anteriormente:

- La función **Busca_Camino_Extension** busca el menor camino partiendo de un vértice concreto (no desde el primero como en **Busca_Camino** de la anterior heurística), es decir calcula y devuelve una par que contiene la combinación de aristas cuya suma total de distancias sea menor y el valor de dicha suma.

```
pair<vector<Arista>,double> Busca_Camino_Extension
    (const vector<vector<Arista>> &matriz, int index){

    pair<vector<Arista>,double> sol;

    vector<Arista> aristas;
    vector<bool> vertices_seleccionados;
    int num_vertices = matriz.size();

    for(int i = 0; i< num_vertices; i++)
        vertices_seleccionados.push_back(false);

    vertices_seleccionados.at(index) = true;
    int pos_actual = index;

    while(aristas.size()<num_vertices){
        double min = 9999999;
        int pos_min = pos_actual;

        for (int i = 0; i < num_vertices; ++i) {
            double d = matriz.at(pos_actual).at(i).distancia;
            if(d<min && d>0 && !vertices_seleccionados.at(i)){
                min = d;
                pos_min = i;
            }
        }

        vertices_seleccionados.at(pos_min) = true;
        aristas.push_back(matriz.at(pos_actual).at(pos_min));
        pos_actual = pos_min;
    }

    aristas.back()= matriz.at(pos_actual).at(index);

    sol.first = aristas;
    sol.second = Suma_Distancias(aristas);

    return sol;
}
```

- La función **Busca_Min_Camino** busca el camino con distancia menor ejecutando la función anterior **Busca_Camino_Extension** para cada uno de los posibles nodos. De todos los posibles resultados, devuelve la mejor combinación de aristas y su suma total.

```
pair<vector<Arista>,double> Busca_Min_Camino
       (const vector<vector<Arista>> &matriz){

    pair<vector<Arista>,double> result;
    result = Busca_Camino_Extension(matriz,0);
    pair<vector<Arista>,double> min = result;

    for(int i=1; i<matriz.size();i++){
        result = Busca_Camino_Extension(matriz,i);

        if(result.second<min.second){
            min.second = result.second;
            min.first = result.first;
        }
    }

    return min;
}
```

10.1. Justificación de la validez

Esta heurística obtiene también una solución factible e incluso mejor, pues usa el algoritmo de la heurística anterior pero optimiza su solución ya que compara todos los posibles caminos y escoge el mejor, es decir el más corto, siempre usando la técnica Greedy de elección en base a criterios locales. Sin embargo, esto supone un coste en eficiencia y en recursos computacionales.

Se especifican a continuación donde se presentan en el algoritmos los elementos propios de la técnica Greedy:

- **Conjunto de Candidatos:** para cada vértice un vector del tipo Arista donde se almacenan las distancias a otros vértices (es cada fila del parámetro **matriz** que se pasa a la función Busca_Camino.
- **Conjunto de seleccionados:** vector del tipo Lista que se devuelve, llamado **aristas**.
- **Función solución:** la función Busca_Min_Camino se queda con el camino de distancia total menor de los encontrados.
- **Función de factibilidad:** se trata de las condiciones para encontrar una nueva arista mínima para poder añadirla al conjunto de soluciones de la función Busca_Camino_Extension. Estas condiciones son que la arista tenga distancia sea estrictamente positiva y que el vértice no haya sido seleccionado.
- **Función de selección:** tomar las arista más pequeña en cada caso y finalmente el camino más corto.

- **Función objetivo:** la función Crea_Camino que se especifica más tarde da el valor de la solución alcanzada, devuelve un vector con las aristas que conforman el camino a seguir.

10.2. Análisis de eficiencia

10.2.1. Análisis teórico

Para analizar la eficiencia teórica de la heurística, estudiemos antes la eficiencia de las funciones que intervienen, en función del número de vértices introducido al que denotamos por n:

1. Función Busca_Camino_Extension: su eficiencia es $\mathcal{O}(n^2)$ igual que **Busca_Camino** pues el cuerpo de la función es igual excepto la asignación en la que se establece el nodo de inicio una operación que sigue siendo constante.

2. Función Suma_Distancias: su eficiencia es $\mathcal{O}(n)$ ya que recorre el vector completo de aristas cuyo tamaño es n y en cada realiza una suma cuya eficiencia es constante.

3. Función Busca_Min_Camino: su eficiencia es $\mathcal{O}(n^3)$ pues ejecutamos n veces la función Busca_Camino_Extension cuya eficiencia era $\mathcal{O}\ (n^2)$.

   En consecuencia, se deduce que la eficiencia teórica es

$$T(n) = \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n^3) = max(\mathcal{O}(n), \mathcal{O}(n^2), \mathcal{O}(n^3)) \in \mathcal{O}(n^3)$$

10.2.2. Análisis empírico

Num. vértices	Tiempo (seg)
100	0.00658327
150	0.0207329
200	0.0199559
250	0.0356111
300	0.0685535
350	0.0779346
400	0.117423
450	0.151521
500	0.238964
550	0.335346
600	0.562435
650	0.562358
700	0.858926
750	0.907555
800	1.03697
850	1.26161
900	1.67673
950	1.69096
1000	1.96148
1050	2.25226
1100	2.55858
1150	2.92032
1200	3.31641
1250	3.72387
1300	4.18349
1350	4.64981
1400	5.12525
1450	5.70853
1500	6.32327
1550	6.95333
1600	7.63414
1650	8.60254
1700	9.09336
1750	9.85312
1800	10.9755
1850	12.456
1900	12.578
1950	13.4856
2000	14.595

Cuadro 4: Tabla del algoritmo PVC3 hasta 2000 puntos con saltos de 50

10.2.3. Análisis híbrido

Como hemos visto en el análisis teórico, el algoritmo es cúbico luego usamos una función polinómica de tercer grado: $T(n) = a_3 \cdot n^2 + a_2 \cdot n^2 + a_1 \cdot n + a_0$. Obtenemos las

constantes ocultas, y así podemos saber el tiempo para cualquier tamaño de entrada n.

Al realizar el ajuste se obtiene para 2000 vértices en saltos de 50, la ecuación $t(n) = 1{,}86345 \cdot 10^{-9} \cdot n^3 + 1{,}73533 \cdot 10^{-9} \cdot n^2 + 4{,}15215 \cdot 10^{-7} \cdot n + 0{,}000314032$ con un coeficiente de determinación de 0,0291195.
Como el residuo es casi cero vemos que el ajuste es muy bueno.

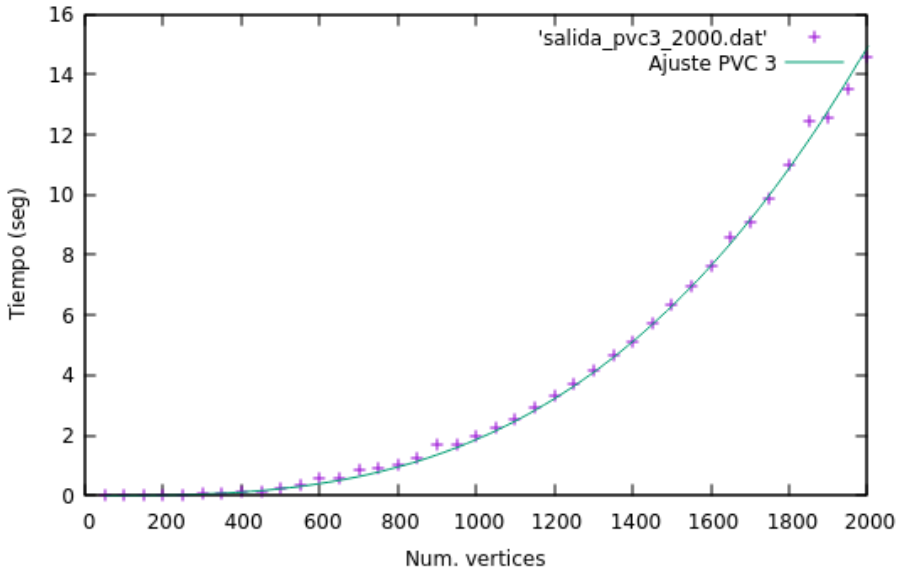

Figura 8: Gráfica del análisis empírico del algoritmo PVC2 para 2000 puntos

Función Crea_Camino

Además, para esta dos últimas heurísticas se ha implementado Crea_Camino, una función que se encarga de devolver una sucesión ordenada de vértices visitados dado un vector de aristas seleccionadas, en definitiva los puntos en orden por los que se desplazará el Viajante de Comercio.

```
vector<int> Crea_Camino(const vector<Arista> &aristas ){
    vector<int> camino_final;

    int num_vertices = aristas.size();

    for (int i=0; i< num_vertices; i++){
        camino_final.push_back(aristas.at(i).p1);
    }

    camino_final.push_back(aristas.back().p2);

    return camino_final;
}
```

11. Análisis comparativo de las tres heurísticas

11.1. Comparación en términos de la calidad de las soluciones

Para comparar las tres soluciones en función de la calidad se han ejecutado las 3 heurísticas para distintos tamaños del problema y con los mismos valores. En la siguiente tabla se muestran los resultados obtenidos.

Además para que sea más sencillo de comparar se presenta a continuación un gráfico con la represenatcion grafica de las soluciones. Está claro que la gráfica

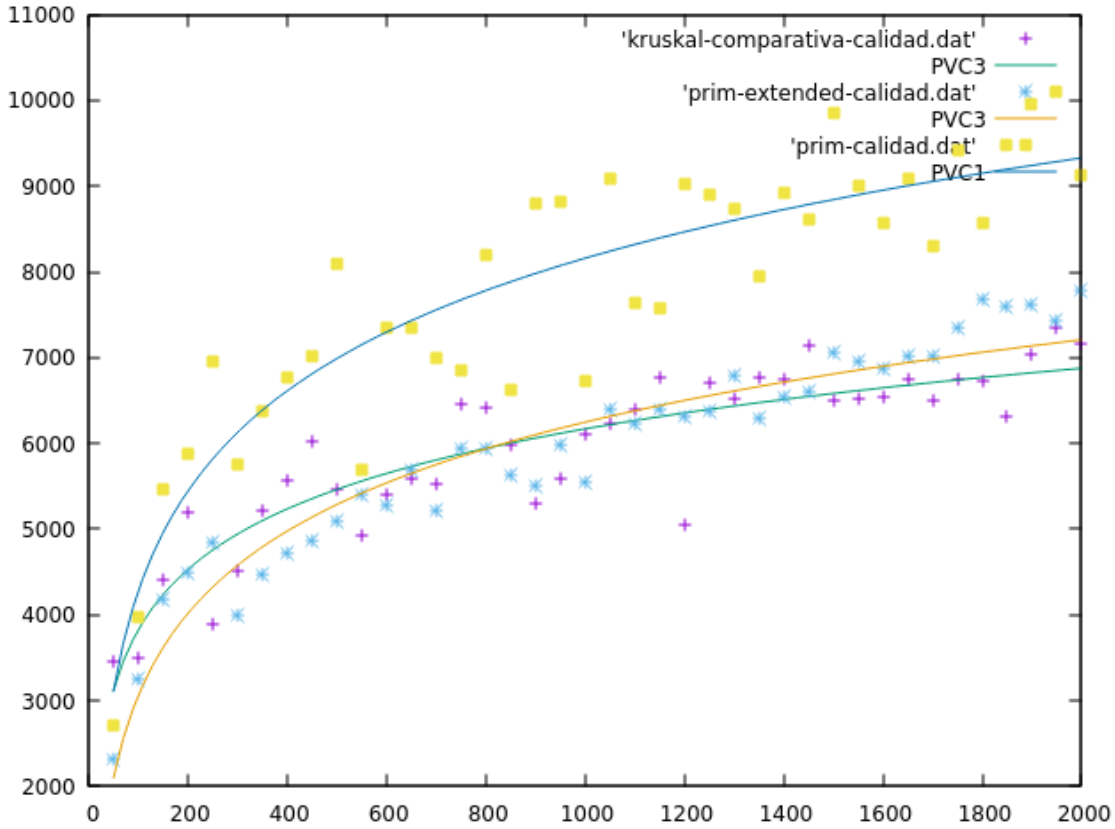

Figura 9: Gráfico comparativo de la calidad de las soluciones

del PVC2 siempre va a estar por debajo de la gráfica del PVC1, puesto que el PVC2 toma la mejor solución que se obtiene de ejecutar el PVC1 partiendo desde diferesntes puntos, luego el PVC2 ofrece soluciones de mayor calidad que el PVC1. Por otro lado, comparando el PVC3, que es el que está basado en el algoritmo de Kruskal, vemos como respecto al PVC2, las soluciones no difieren tanto. En algunos puntos el PVC3 ofrece soluciones mejores que el PVC2 Y viceversa, luego en cuestión de calidad lo idóneo sería elegir el PVC3.

	PVC1	PVC2	PVC3
50	2705.62	2319.95	3466
100	3970.84	3255.99	3501
150	5472.32	4172.94	4411
200	5878.14	4497.99	5204
250	6955.57	4840.63	3888
300	5744.66	4001.63	4517
350	6370.96	4464.35	5209
400	6765.05	4711.39	5565
450	7017.52	4868.86	6025
500	8099.04	5095.24	5473
550	5684.2	5399.68	4935
600	7338.9	5278.62	5398
650	7348.47	5697.08	5582
700	6996.2	5217.89	5528
750	6841.63	5933.9	6462
800	8187.79	5933.14	6420
850	6625.76	5632.05	5991
900	8791.67	5505.34	5306
950	8827.29	5979.64	5581
1000	6728.47	5548.53	6099
1050	9086.32	6391.65	6222
1100	7641.38	6235.86	6405
1150	7565.81	6403.23	6773
1200	9016.55	6319.2	5051
1250	8900.71	6376.29	6706
1300	8732.23	6785.64	6519
1350	7948.91	6284.53	6759
1400	8917.3	6546.21	6750
1450	8606.43	6606.27	7134
1500	9842.83	7068.24	6498
1550	9006.62	6945.19	6526
1600	8574.59	6877.89	6547
1650	9083.62	7026.64	6746
1700	8301.25	7011.48	6508
1750	9423.45	7339.47	6748
1800	8568.79	7669.33	6719
1850	9484.93	7590.85	6319
1900	9961.98	7624.59	7043
1950	10107.4	7436.77	7343
2000	9126.85	7776.32	7169

Cuadro 5: Tabla comparativa de la calidad de las soluciones

11.2. Comparación en términos del tiempo de ejecución

A continuación se presenta una tabla comparativa de los tiempos de ejecución de las tres heurísticas en función de los diferentes tamaño del problema.

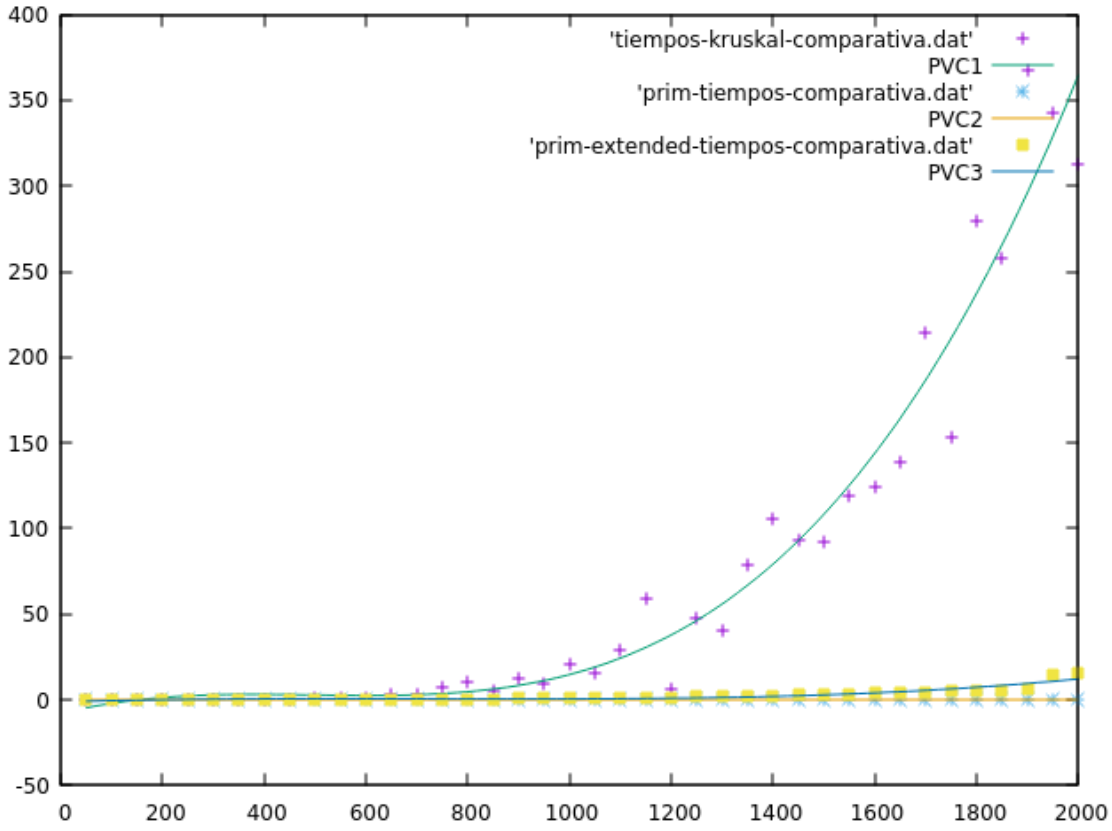

Figura 10: Gráfico comparativo de los tiempos de ejecución

En lo relativo a la eficiencia en tiempos de ejecución de cada una de las tres heurísticas queda claro, en vista de los datos expuestos en la tabla y de la comparación visual que permite la gráfica, que la primera heurística -la que se basa en Kruskal- tiene los peores tiempos de ejecución, pues aunque la tercera heurísticas también tenga una eficiencia cúbica tiene unas constantes muchos menores, que en comparación con las del PVC1 las hace insignificantes. En lo relativo al PVC2 Y al PVC3, se parecía como el PVC3 tiene tiempos de ejecución de mayores lo cual concuerda con su implementación, pues calcula todas las soluciones posibles entre ellas la del PVC2 y toma la menor. La tercera heurística tiene una eficiencia cúbica y la segunda la tiene cuadrática, aunque en comparación con las primera podrían ser casi de la misma eficiencia. Por todo esto, la mejor implementación en relación al tiempo es la segunda.

	PVC1	PVC2	PVC3
50	2.4699e-05	0.000832327	0.00244559
100	6.1631e-05	0.00497175	0.00470864
150	3.6331e-05	0.0071572	0.0212251
200	6.7673e-05	0.00998897	0.0648226
250	9.7262e-05	0.0185916	0.0463576
300	0.000117446	0.0302149	0.156311
350	0.000200139	0.0452752	0.645985
400	0.000214173	0.0669002	0.543095
450	0.000241994	0.0932323	0.624432
500	0.000309993	0.125929	0.923133
550	0.000350322	0.155565	1.07908
600	0.000454122	0.204324	1.09428
650	0.000479444	0.254967	2.74742
700	0.000611671	0.312296	3.61451
750	0.000646074	0.38575	7.7536
800	0.000984067	0.464391	10.0575
850	0.000861369	0.544774	5.61867
900	0.000911462	0.657452	12.9607
950	0.0010267	0.776551	9.28306
1000	0.00110805	0.880465	20.5682
1050	0.00121036	1.01628	15.6475
1100	0.00144453	1.23558	29.2588
1150	0.00143746	1.33875	59.4058
1200	0.00159027	1.54567	6.6205
1250	0.00164265	1.75772	48.1589
1300	0.00185073	1.99652	40.7835
1350	0.00184842	2.23603	78.6503
1400	0.00199282	2.54266	105.524
1450	0.00217164	2.82047	93.5514
1500	0.00226821	3.18296	91.904
1550	0.00243763	3.53082	119.092
1600	0.00255108	3.85008	124.774
1650	0.00269964	4.29472	138.633
1700	0.00277235	4.54496	214.58
1750	0.00298053	4.96893	153.198
1800	0.00315019	5.57299	141.605
1850	0.0032008	5.74671	68.7349
1900	0.00353015	6.27421	370.283
1950	0.00838644	14.4942	342.995
2000	0.00824375	15.4593	454.251

Cuadro 6: Tabla comparativa de los tiempos de ejecución

12. Conclusiones

El desarrollo de esta práctica nos ha permitido ver la eficacia que presentan los algoritmos greedy a la hora de resolver ciertos problemas, como nos permiten alcan-

zar en numerosas ocasiones soluciones óptimas, con la ventaja de proporcionar un algoritmo **sencillo, fácil de implementar y rápido(eficientes)**.

En el caso del **problema del catering** se ha podido aplicar la filosofía de los algoritmos voraces de buscar siempre la mejor opción en cada momento, y en este caso(no siempre va a ocurrir) da una solución óptima. Hemos visto la importancia de estudiar **la corrección del algoritmo** para verificar su optimalidad.

Sin embargo, también hemos visto como **hay ciertos problemas en los que esta práctica no garantiza la solución óptima**. Otros ejemplos de ellos serían el problema de la mochila, el problema del coloreo de grafos, entre otros. En particular, en el caso del problema del viajante de comercio, se ha visto y explicado como **cada heurística presenta beneficios distintos** si hablamos de tiempo de ejecución o calidad de las soluciones. Se ha expuesto como el **PVC2**, es **el mejor en cuanto a velocidad**, mientras que el **PVC3**, **normalmente ofrece la mejor solución**. A la hora de elegir una de las tres heurísticas sería interesante hacerlo en cuanto a relación calidad/tiempo, y en tal caso **la elegida sería la tercera**, como se explicó en el apartado anterior tiene la mejor solución y su diferencia en tiempo con respecto a PVC2 es mínima, teniendo en cuenta que PVC2 no encuentra siempre la menor solución.