UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2022 - 2023

Algorítmica

Memoria de prácticas

Práctica 2: Greedy(Algoritmos voraces)

Autores:

- Mario Martín Rodríguez mario10@correo.ugr.es
- Juan Ayuso Arroyave juanayuso@correo.ugr.es
- Mario Líndez Martínez mariolindez@correo.ugr.es

Índice

1. Autores:

- Mario Líndez Martínez: 33’33%
- Mario Martín Rodríguez: 33’33%
- Juan Ayuso Arroyave: 33’33%

Las tres heurísticas se han desarrollado entre los tres miembros del grupo.

El PowerPoint ha sido realizado por Mario Martín.

La Eficiencia Teórica ha sido realizada por Mario Líndez.

Del resto de la eficiencia se ha encargado Juan Ayuso.

La resolución del problema de la Jukebox fue implementado por Mario Martín y por Mario Líndez.

2. Objetivos

El objetivo de esta práctica es el uso y compresión del algoritmo voraces sobre. Para ello, en primer lugar, hemos diseñado un algoritmo que resuelve el problema de la Jukebox para aprender su utilización correcta.

Además, tendremos que aprender a trabajar con un problema NP-duro, en este caso el problema del viajante de comercio, los cuales son problemas para los que no existe un algoritmo conocido que puede garantizar encontrar la solución óptima en todos los casos.

3. Definición del problema:

3.1. Enunciado

Para animar la estancia de los clientes de un centro comercial se ha instalado un reproductor de música, jukebox, en la que los clientes, a cambio de 10 céntimos pueden seleccionar una canción de entre los clásicos más populares para que se reproduzca. El encanto de este dispositivo es que la música se haya guardada en vinilos, no en soporte digital. El sistema tiene un amplio repertorio de música, pero organizado en dos grupos. Hay un primer grupo de vinilos cuya reproducción es inmediata. Es decir, en cuanto se selecciona la canción se puede reproducir. Sin embargo, el espacio disponible para este grupo de vinilos es limitado. El resto de vinilos están en un espacio adjunto.

Si la canción que selecciona el cliente no está entre los vinilos disponibles para reproducción inmediata, se puede reproducir, pero antes hay que reemplazar uno de los vinilos en el grupo de disponibilidad inmediata por el vinilo que incluye la canción seleccionada. Supongamos que el número total de canciones en la caja de música es c, y que el número máximo de canciones que se pueden incluir en el grupo de reproducción inmediata es k. Supongamos que de partida no hay ningún vinilo en este grupo de reproducción inmediata y que se irá rellenando con las peticiones de los clientes. Por simplicidad, asumamos que cada vinilo incluye sólo una canción. Se pide diseñar un algoritmo para, dada una secuencia de peticiones de canciones, minimizar el número de reemplazos de vinilos dentro del grupo de reproducción inmediata.

Esto se traduce de la siguiente manera:

Contamos con una lista de peticiones de canciones, las cuales van asignadas a un ID (concretamente un número). Queremos ver cuáles de ellas van a meterse en la cola de reproducción inmediata de tal manera que el número de cambios que hagamos sea el menor posible. Esto es manejar una memoria caché conociendo con antelación las peticiones.

3.2. Entorno de análisis

El análisis se realizará desde el ordenador de Juan Ayuso Arroyave, cuyas especificaciones son las siguientes:

- Nombre: HP Laptop 15s-eq2xxx
- Procesador: AMD® Ryzen 7 5700u with radeon graphics × 16
- Disco Duro Sólido (SSD) de 1 TB
- Memoria RAM: 16 GB
- Sistema Operativo: Ubuntu 22.04.1 LTS de 64 bits
- Compilador: g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

3.3. Medición de tiempos

Para medir los tiempos hemos usado la biblioteca de la STL chrono, la cual permite calcular los tiempos de ejecución de forma más precisa. La implementación es la siguiente:

```
#include <chrono>
using std::chrono;

high_resolution_clock::time_point t_antes, t_despues;
duration<double> transcurrido;
t_antes = high_resolution_clock::now();
//sentencia o programa a medir
t_despues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>> (t_despues - t_antes);
cout << "el tiempo empleado es " << transcurrido.count() << " s." << endl;
```

4. Algoritmo Diseñado:

4.1. Realización:

Para la resolución del problema de la Jukebox, haremos uso de un vector de reproducción inmediata y otro vector de peticiones, con las canciones que el cliente desea reproducir. De forma arbitraria, hemos dictaminado que el tamaño del vector de reproducción inmediata sea una quinta parte del tamaño del vector de peticiones. Así si se realizan N peticiones el vector inmediato será de tamaño $\frac{N}{5}$. Nuestro objetivo es introducir todos los elementos del vector peticiones en el vector inmediato de manera que se realicen el mínimo número de cambios posibles.

Para ello hemos planteado un algoritmo, el cual tiene una complejidad O(n^3). El algoritmo consiste en lo siguiente:

En primer lugar, comprobamos si la canción que vamos a introducir del vector de peticiones se encuentra ya en el vector inmediato y en caso afirmativo, no tendremos que realizar ningún intercambio. Para esto nos ayudaremos de la siguiente función:

```
bool Pertenece(const vector<int> & v,int n){            // O(n)

    bool pertenece=false;
    int contador=0;

    while(!pertenece && contador<v.size()){

        if(v.at(contador)==n)
            pertenece=true;

        contador++;
    }
```

```
13.
14.     return pertenece;
15. }
```

Como podemos comprobar posee una eficiencia de O(n) ya que se realizan n comprobaciones.

En caso de que la canción que vamos a introducir no se encuentre en el vector inmediato calcularemos el índice de canción del vector inmediato que se encuentre "más lejos" en el vector de peticiones, o lo que es los mismo, la que se introducirá más tarde (si una canción no se ha vuelto a pedir más tendrá una distancia de N+1). De manera que cambiaremos la canción más lejana del vector inmediato por la canción a introducir del vector peticiones. Para ello nos apoyaremos en la función CanciónMásLejana:

```
 1. int CancionMasLejana (const vector<int> & inmediato, const vector<int> & peticiones, int
pos_inicial){
 2.     int indice = 0;
 3.     int max_distancia = 0;
 4.     int contador = 0;
 5.
 6.     for (int i = 0; contador != peticiones.size() && i < inmediato.size(); ++i){
 7.         bool encontrado = false;
 8.         contador = pos_inicial;
 9.
10.         while (contador < peticiones.size() && !encontrado){
11.             if (inmediato.at(i) == peticiones.at(contador)){
12.                 encontrado = true;
13.             } else {
14.                 ++contador;
15.             }
16.         }
17.
18.         if (contador > max_distancia){
19.             max_distancia = contador;
20.             indice = i;
21.         }
22.     }
23.     return (indice);
24. }
```

Esta función calculará la distancia de cada canción del vector inmediato con su posición en el vector de peticiones a partir de una posición inicial y buscará la canción que más tiempo tarde en volver a repetirse. Para que sea algo más eficiente, se determinará como la más lejana aquella canción que no se encuentre en el vector de posiciones, parando el cálculo.

Esta función al tener que recorrer el vector de peticiones tantas veces como canciones posea el vector inmediato, en el peor caso se tendrán que realizar $\frac{N}{5} * N$ iteraciones, luego es de Orden $O(n^2)$.

Con todo esto, nuestro algoritmo será así:

```
 1. for (int i = 0; i < NUM_PETICIONES; ++i){
 2.     if (!Pertenece(inmediato, peticiones.at(i))){ // Pertenece = O(N)
 3.         if (inmediato.size() < TAMANIO_INMEDIATO){
 4.             inmediato.push_back(peticiones.at(i));
 5.
 6.         } else {
 7.             inmediato.at(CancionMasLejana(inmediato, peticiones, i)) = peticiones.at(i);
 8.             ++NUM_INTERCAMBIOS;
 9.         }
10.
11.     }
12. }
```

Buscaremos primero si el punto se encuentra ya en el vector inmediato, si es así no se hará nada. En caso contrario insertaremos directamente la canción en el vector si éste todavía no se ha llenado ó la intercambiaremos por la canción más lejana.

4.2. Justificación de la validez del algoritmo:

El objetivo de nuestro algoritmo es minimizar el número de reemplazos de canciones. Sean:

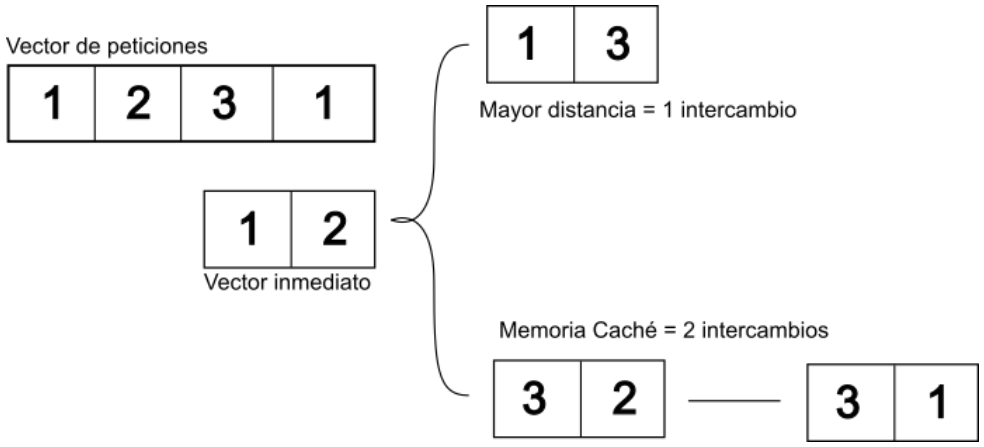
$$S_1, S_2, \dots, S_n$$

las n canciones a introducir en el vector inmediato y sea $h = \frac{n}{5}$ número de huecos del vector inmediatos.

En el caso de que todas las canciones del vector de peticiones estén en el vector inmediato, no habrá que realizar ningún intercambio. Esto es que en el vector de peticiones haya tantas canciones diferentes como tamaño tiene el vector inmediato. Por tanto, es óptimo al producirse el mínimo número de intercambios.

En caso contrario, es decir que haya canciones del vector de peticiones que no se encuentren en inmediato, distinguiremos entre si el vector inmediato posee espacio disponible todavía para insertar una nueva canción, en cuyo caso no se produce ningún intercambio, o si no buscamos la canción del vector inmediato que se encuentre más alejada en el vector de peticiones.

Supongamos que en el vector de peticiones tenemos h+1 canciones diferentes, veamos que ocurre en el momento en el que h canciones se encuentren en el vector inmediato. Al intentar añadir la canción h+1 se producirá un intercambio. Siendo la canción S_k la canción con la mayor distancia, si no la escogemos para realizar quitarla del inmediata se producirá otro intercambio en un menor tiempo en comparación a si sí la hubiéramos escogido. Por tanto, podemos concluir que escoger S_k es la que produce menor número de intercambios.



5. Eficiencias:

5.1. Eficiencia teórica:

Si analizamos la eficiencia de este bucle con todo lo anterior descrito podemos ver que se realizaran $\frac{N}{5}$ iteraciones. Dentro del cuerpo del bucle nos encontramos con la primera condición, (O(n)),más adentro tenemos un if-else cuya eficiencia será la mayor entre las partes, siendo inmediato en el caso if y $O(N^2)$ en el else. Por tanto, el cuerpo del bucle será $\frac{N}{5} * \left(\frac{N}{5} + N^2\right) = N^3 + \frac{N^2}{10} \in O(n^3)$.

5.2. Eficiencia empírica:

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Para este algoritmo, lo ejecutaremos para diferentes tamaños del vector de la lista de canciones. Estos tiempos los almacenamos en un fichero jukebox.dat. El código es el siguiente:

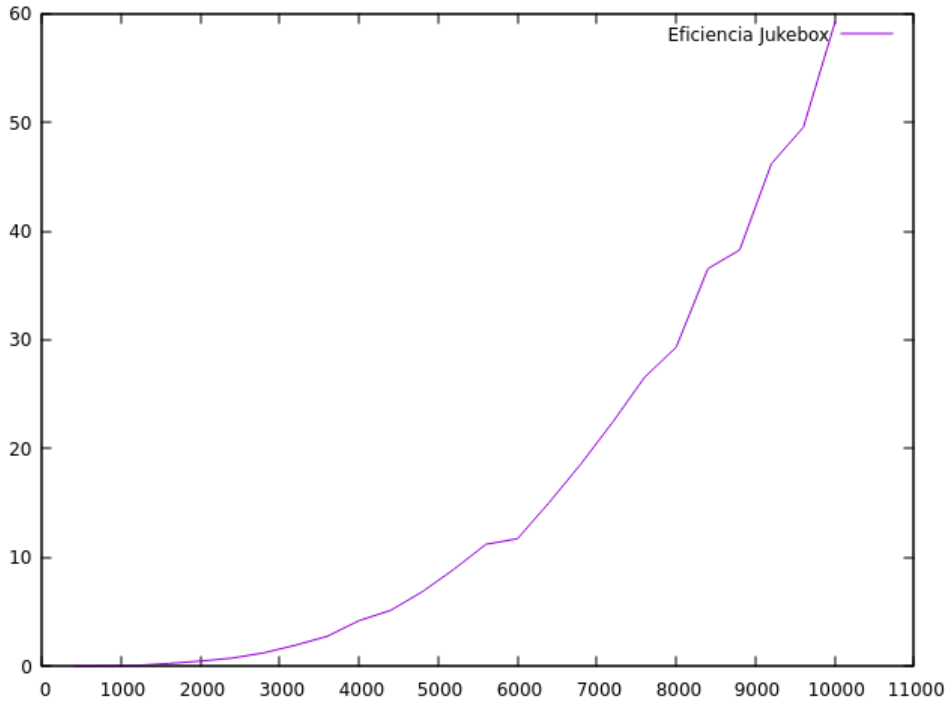
```
#!/bin/bash
pinté "" > jukebox.dat

i=400
while [ "$i" -le 10000]
do
# Generamos los puntos
./generador $i

# Ejecutamos los puntos
./voraz data_jukebox.txt >> jukebox.dat

echo "Terminado $i"

i=$(( $i + 400))
done
```

Empezamos con un tamaño base de 400 puntos y vamos aumentándolo de 400 en 400 hasta llegar al tamaño de 10000 puntos.

Los tiempos obtenidos son los siguientes:

Tamaño	Tiempo (seg)
400	0,0062
800	0,0319
1200	0,0921
1600	0,2779
2000	0,5010
2400	0,7707
2800	1,2595
3200	1,9579
3600	2,7711
4000	4,2045
4400	5,1643
4800	6,8767
5200	8,9378
5600	11,2105
6000	11,7428
6400	15,0724
6800	18,6077
7200	22,4587
7600	26,5604
8000	29,3323
8400	36,5494
8800	38,2621
9200	46,1825
9600	49,5025
10000	59,1766

A partir de dichos tiempos, queda la siguiente gráfica:



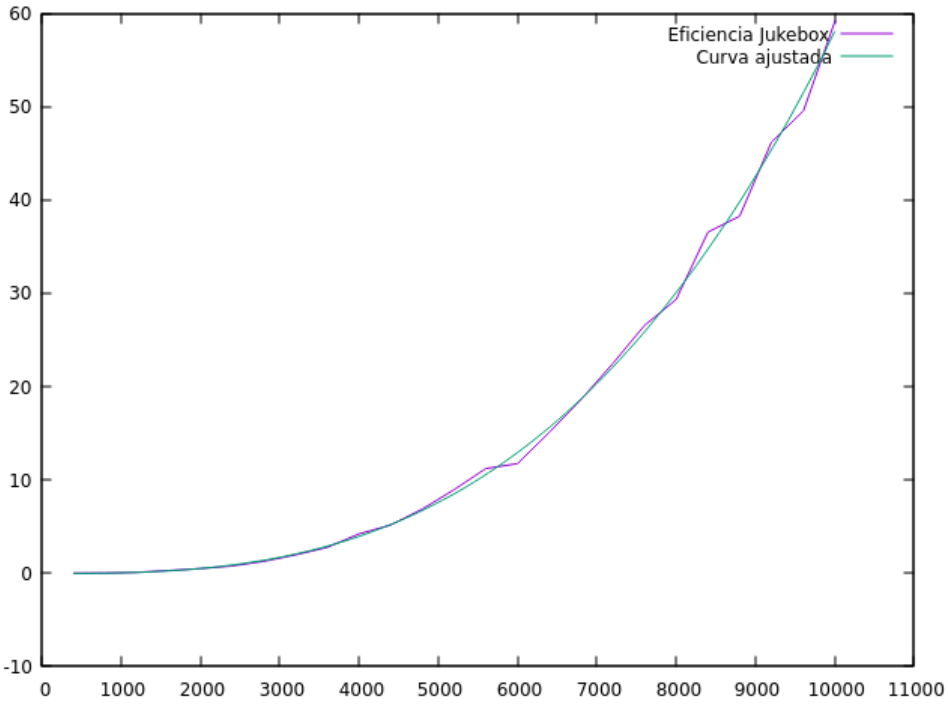
Gráfica de tiempos del algoritmo diseñado

5.3. Eficiencia híbrida:

Para la eficiencia híbrida, hemos realizado un ajuste con la función $f(x) = a_0 x log(x)$ y como resultado hemos obtenido las siguientes constantes ocultas:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a3              = 5.45345-11       +/- 9.09e-12   (16.67%)
a2              = 3.79601e-08      +/- 1.437e-07  (378.5%)
a1              = -3.33517e-05     +/- 0.0006498  (1948%)
a0              = -0.0207235       +/- 0.7962     (3842%)
```

Cuyo coeficiente de correlación es de 0,9233 por lo que el ajuste es muy bueno. Nos queda finalmente la curva $f(x) = 5'45 \cdot 10^{-11} x^3 + 3'8 \cdot 10^{-8} - 3'34 \cdot 10^{-5} - 0'02$



6. Primera heurística del PVC

6.1. Realización:

```
 1. while (visita.size() < NUM_NODOS){
 2.         min_dist = 500;
 3.         for (int i = 1; i < NUM_NODOS; ++i){
 4.             int num = distancias[pos_actual][i];
 5.
 6.             if((!ya_pertenece[i]) && (num < min_dist) && (i != pos_actual)){
 7.                 min_dist = num;
 8.                 indice_minimo = i;
 9.             }
10.         }
11.         distancia_solucion += min_dist;
12.
13.         pos_actual = indice_minimo;
14.         visita.push_back(pos_actual);
15.         ya_pertenece[pos_actual] = true;
16.     }
17.
18.     distancia_solucion += distancias[visita.at(0)][visita.at(visita.size()-1)];
19.
```

Para esta primera heurística, hemos realizado lo siguiente:

En primer lugar, generamos una matriz con las distancias que hay entre cada punto. La diagonal de dicha matriz no la vamos a tener en cuenta ya que va a ser la distancia de un punto consigo mismo. Creamos también un vector que va a guardar el orden en el que debemos ir para que la distancia sea mínima, empezando por la fila 0 que es la empresa.

Comenzamos a partir de la fila que representa a la empresa, y vamos a buscar qué columna de dicha fila guarda el menor valor, es decir, una menor distancia. Una vez encontrado, la siguiente fila que vamos a comprobar va a ser la que corresponde con la columna que tenía el menor valor. Además, añadimos la fila al vector que guarda el orden y a través de un bool creado previamente, vamos a marcar dicha fila como ya perteneciente para así no tenerla en cuenta en el resto de las iteraciones.

6.2. Justificación de su validez:

Nuestro problema del viajero es del tipo NP duro, lo cual significa que no hay ningún algoritmo conocido que pueda garantizar encontrar la solución óptima en todos los casos.

Sabemos, por tanto, que nuestra solución no va a ser óptima. Sin embargo, en este caso nuestra heurística se basa en el algoritmo de Prim, ya que buscamos el vecino más cercano, y dicho algoritmo es óptimo en determinados casos.

7. Segunda heurística del PVC

7.1. Realización:

Para esta segunda heurística nos hemos basado en el ya conocido algoritmo de Kruskal. En él iremos buscando las aristas de menor peso y seleccionando los nodos que unen considerando siempre que no se deban formar ciclos. Además, para su uso en este problema, tendremos que tener en cuenta el número de "conexiones" que posee cada nodo, pues el viajero solo deberá llegar a un nodo k e ir al siguiente, sin posibilidad de volver. Para esto iremos llevando la cuenta de cuantos caminos posee cada nodo.

Para la implementación de nuestro algoritmo nos apoyaremos del struct Arista, el cual poseerá sobrecargado el operator<, que lo usaremos para ordenar el vector de aristas dentro del main según sus pesos. Esta ordenación vendrá dada por la función qsort implementada por la stl.

```
1. struct Arista {
2.     int origen;
3.     int destino;
4.
5.     int peso;
6.
7.     bool operator< (const Arista & otro) const {
8.         return(this->peso > otro.peso);
9.     }
10.
11. };
```

Ya en el programa principal, usaremos un array de enteros que llamaremos caminos el cual nos irá indicando cual es el nodo siguiente por visitar, es decir, si $caminos[2] = 3$ significará que estarán conectados el nodo 2 y 3 siendo ese el camino a seguir. Si un nodo cierra el camino estos valores serán coincidentes, esto es $caminos[k] = k$, por lo que lo llamaremos "nodo raíz".

Veamos el código del algoritmo:

```
1. while (solucion.size() < NUM_NODOS-1){
2.     Arista siguiente_arista = aristas.back();
3.     aristas.pop_back();
4.
5.     int origen = siguiente_arista.origen;
6.     int destino = siguiente_arista.destino;
7.
8.     // Cada nodo solo podrá tener dos conexiones
9.     if ((num_conexiones[origen] < 2) && (num_conexiones[destino] < 2)){
10.        int raiz_a = BuscaRaiz(caminos, origen);    // O(n)
11.        int raiz_b = BuscaRaiz(caminos, destino);   // O(n)
12.
13.        if (raiz_a != raiz_b){
14.            solucion.push_back(siguiente_arista);
15.            caminos[origen] = destino;
16.            ++num_conexiones[origen];
17.            ++num_conexiones[destino];
18.        }
19.
20.    }
21. }
```

Su funcionamiento es el siguiente:

Realizaremos tantas iteraciones como aristas necesitemos, esto es N-1. Sacaremos del vector de aristas ordenado la arista que tenga menor peso actualmente y comprobaremos si el número de conexiones del origen o del destino es menor que 2. En caso de que así sea, se comprobará si se forman ciclos buscando la raíz del nodo y viendo si son coincidentes, esto es ver si el nodo origen y el nodo destino ya se encuentran conectados de alguna manera. Si no forman parte de un camino, los uniremos añadiendo la arista correspondiente al vector solución y actualizando la información relativa a los nodos (indicamos el camino formado y sumamos uno al contador de conexiones de los dos).

Al final de todo nos faltará unir el primer y último nodo del camino, por lo que con una rápida pasada al vector de conexiones identificaremos los nodos faltantes para cerrar el camino.

```
// Buscamos la ultima conexion faltante para completar el ciclo
bool done = false;
int i = 0;
while (!done){  // O(n)
    if (num_conexiones[i] == 1){
        int j = i+1;
        while (!done){
            if (j != i && num_conexiones[j] == 1){
                solucion.push_back(Arista{j,i,distancias[i][j]});
                done = true;
            }
            ++j;
        }
    }
    ++i;
}

```

Con todo, este algoritmo será del orden que sea el algoritmo de ordenación que utilicemos, en este caso al ser el Quicksort, $O(nlog_2(n))$.

7.2. Justificación de su validez:

Nuestro problema del viajero es del tipo NP duro, lo cual significa que no hay ningún algoritmo conocido que pueda garantizar encontrar la solución óptima en todos los casos.

Sabemos, por tanto, que nuestra solución no va a ser óptima. Sin embargo, en este caso nuestra heurística se basa en el algoritmo de Kruskal, ya que buscamos la arista de menor peso, y dicho algoritmo es óptimo en determinados casos.

8. Tercera heurística del PVC

8.1. Realización:

Para realizar la tercera heurística:

En primer lugar, generamos una matriz con las distancias que hay entre cada punto. La diagonal de dicha matriz no la vamos a tener en cuenta ya que va a ser la distancia de un punto consigo mismo.

Creamos también un vector solución que va a guardar el orden en el que debemos ir para que la distancia sea mínima.

Para rellenar dicho vector, vamos a comenzar buscando cual es el punto cuya distancia al 0 es menor y vamos a introducir el 0 en la primera posición del vector solución (salimos de la empresa) y el punto con menor distancia al 0 en la última posición del vector solución.

```
for(int i = 1; i < NUM_NODOS; ++i){
    if(distancias[0][i] < minimo){
        minimo = distancias[0][i];
        p1 = i;
```

```
    }

}

visita.at(0) = 0;                    // metemos 0 al principio salimos de la empresa
visita.at(NUM_NODOS-1)=p1;          // metemos p1 al final que es el
// ya sabemos que la minima distancia es minimo y que se produce entre los puntos p1 y 0
// vamos a coger el nodo k que tenga menor cantidad dist(p1,k) + dist(0,k) - dist(p1,0)
ya_pertenece[0] = true;
ya_pertenece[p1] = true;

```

Para el resto de los puntos, tendremos que buscar el punto p que cumpla que su distancia al 0 más su distancia al último punto del vector solución sea menor. Una vez encontrado dicho punto p, entonces introduciremos p en la última posición disponible del vector, lo más a la derecha del vector posible, y p pasará a ser el último punto con el que calculemos las distancias.

```
while(contador < NUM_NODOS-1){
    int suma_minima=200;
    for (int i = 1; i < NUM_NODOS; ++i){
        int num = distancias[i][p1] + distancias[i][0] - minimo;
        if((!ya_pertenece[i]) && (num < suma_minima) && (i != p1)){
            suma_minima=num;
            indice=i;

        }
    }
    visita.at(NUM_NODOS-contador-1)=indice;  // lo metemos en la última posición no ocupada
    ya_pertenece[indice] = true;

    p1=indice;
    distancia_solucion += suma_minima;
    contador++;
}

```

8.2. Justificación de su validez:

Nuestro problema del viajero es del tipo NP duro, lo cual significa que no hay ningún algoritmo conocido que pueda garantizar encontrar la solución óptima en todos los casos.

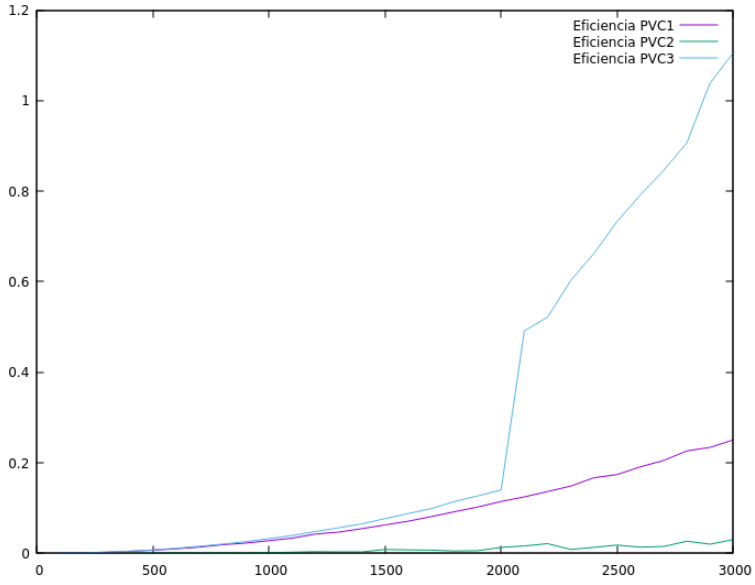
Sabemos, por tanto, que nuestra solución no va a ser óptima. Sin embargo, en este caso nuestra heurística, al igual en la primera heurística, se basa en el algoritmo de Prim, ya que buscamos el vecino más cercano, con la particularidad de que es el vecino más cercano a dos puntos en lugar de a uno y dicho algoritmo es óptimo en determinados casos.

9. Análisis comparativo de las tres heurísticas

Tamaño	Tiempo (segundos)		
	Pvc1	Pvc2	Pvc3
100	0,00070	0,00003	0,00096
200	0,00115	0,00018	0,00124
300	0,00259	0,00017	0,00281
400	0,00460	0,00034	0,00503
500	0,00707	0,00063	0,00790
600	0,01020	0,00144	0,01138
700	0,01392	0,00103	0,01569
800	0,01908	0,00137	0,02061
900	0,02292	0,00105	0,02604
1000	0,02810	0,00178	0,03271
1100	0,03385	0,00277	0,03965

1200	0,04269	0,00388	0,04832
1300	0,04721	0,00343	0,05670
1400	0,05460	0,00338	0,06536
1500	0,06285	0,00871	0,07675
1600	0,07126	0,00778	0,08801
1700	0,08133	0,00735	0,09914
1800	0,09206	0,00534	0,11469
1900	0,10245	0,00598	0,12709
2000	0,11472	0,01347	0,14025
2100	0,12476	0,01659	0,49151
2200	0,13654	0,02160	0,52179
2300	0,14872	0,00875	0,60290
2400	0,16707	0,01348	0,66251
2500	0,17432	0,01858	0,73323
2600	0,19133	0,01389	0,79192
2700	0,20524	0,01536	0,84598
2800	0,22643	0,02685	0,90696
2900	0,23415	0,02035	1,03956
3000	0,25065	0,03062	1,10728

Esta es la tabla de tiempos de las heurísticas donde vemos diferencias significativas de tiempos entre ellas. Observemos la gráfica comparativa para más claridad:



Comparación de la eficiencia de las tres heurísticas (segundos)

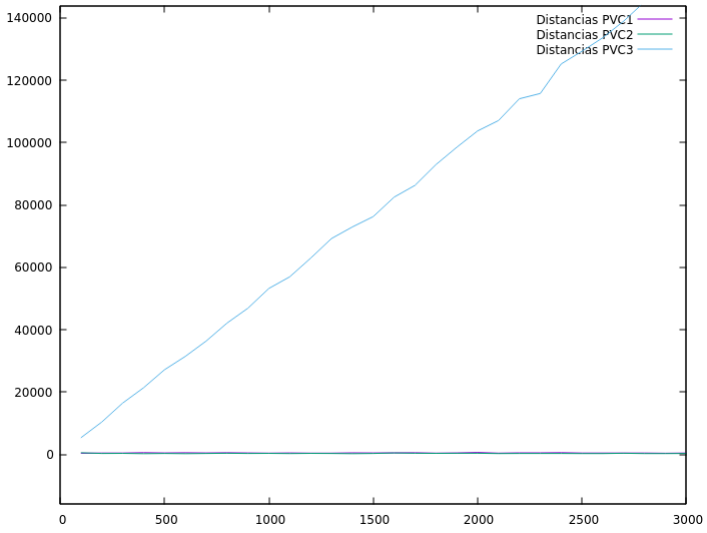
Determinamos claramente cómo la heurística dos es la más eficiente de todas, esperable de un $O(nlog(n))$.

Antes de determinar cual es la "mejor" veamos antes la calidad de sus soluciones:

Tamaño	Tiempo (segundos)		
	Pvc1	Pvc2	Pvc3
100	387	539	5375
200	444	328	10408
300	456	345	16496
400	577	266	21406
500	494	330	27144

600	551	256	31474
700	490	329	36382
800	547	390	42161
900	475	313	46931
1000	431	356	53250
1100	494	304	56991
1200	407	345	62951
1300	421	332	69256
1400	509	250	72987
1500	493	326	76293
1600	529	476	82510
1700	535	389	86326
1800	425	355	92933
1900	470	388	98518
2000	649	389	103760
2100	407	300	107100
2200	510	309	114079
2300	511	320	115789
2400	585	318	125228
2500	461	305	129349
2600	455	296	133696
2700	457	378	139009
2800	464	284	145186
2900	399	285	148358
3000	454	316	147602

Veamos la gráfica comparativa:



Comparación en calidad de solución

Observamos como la heurística 1 y 2 ofrecen soluciones similares, sin embargo, la tercera ofrece muy malas soluciones.

Por tanto, podemos concluir que la tercera heurística es claramente la peor de todas, tanto en tiempo como en calidad de soluciones. También cabe destacar que, a pesar de ofrecer soluciones de calidad similar, la heurística dos es la mejor de todas por su gran eficiencia.

10. Conclusión

En cuanto al algoritmo Greedy, concluimos que a pesar de ser algoritmos fáciles de pensar e implementar, su eficiencia no siempre es la mejor de todas, llegando a órdenes muy poco factibles tales como un $O(n^3)$.

En lo relacionado a las heurísticas, hemos visto que diseñarlas de forma correcta es primordial ya que, si esta no es buena se nos podrán disparar tanto el tiempo de ejecución como la pobreza en sus soluciones respecto a una mejor diseñada.