Práctica 3
Algorítmica

Algoritmos Greedy (Voraces)

Realizado por grupo Orquídeas (subgrupo A1):

- Azorín Martí, Carmen
- Cribillés Pérez, María
- Ortega Sevilla, Clara
- Torres Fernández, Elena

Profesora: Lamata Jiménez, María Teresa

Curso 2021/2022 2º cuatrimestre

Índice

1. Introducción

En esta práctica hemos practicado el aplicar el enfoque greedy para resolver un problema y así crear un algoritmo Greedy. Esta técnica se basa en seleccionar en cada caso lo mejor entre los candidatos, sin tener en cuenta lo ya hecho. La primera solución suele ser la óptima, pero no tenemos la garantía, por tanto, hemos estudiado la corrección del algoritmo para ver si era la óptima o no.

Primero de todo hemos estudiado si se puede resolver el problema con esta técnica viendo 6 características (suficientes, no necesarias):

- Hay un **conjunto de candidatos** para ejecutar una tarea
- Hay una **lista de los candidatos** que ya se han **usado**
- Una **función solución** para estudiar cuando un conjunto de candidatos forma una solución
- **Criterio** que estudia si es **factible** el conjunto de candidatos (puede ser solución)
- Hay una **función de selección** para seleccionar el candidato perfecto en cada instante (de los no usados)
- Hay una **función objetivo** que a cada solución le asocia un valor: intentamos optimizar

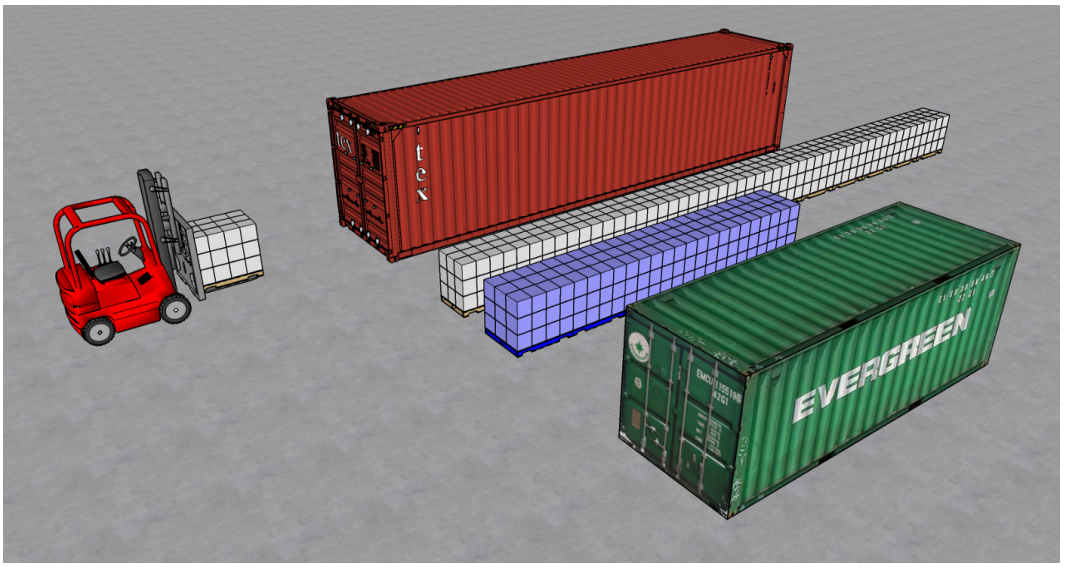
El pseudocódigo de esta técnica podría ser el siguiente:

```
S=∅
Mientras S no sea una solución y C≠∅. Hacer:
        X=elemento de C que maximiza SELEC(X)
        C=C-{X}
        Si (S U {X}) es factible entonces S=SU{X}
Si S es una solución entonces devolver S
en caso contrario: no hay sol
```

C = lista candidatos
S = solución
SELEC(X) = función selección

2. Ejercicio Contenedores

En este ejercicio tenemos un buque de capacidad K toneladas y $c_1, c_2, ..., c_n$ contenedores con sus respectivos pesos $p_1, p_2, ..., p_n$. Hemos diseñado un algoritmo que maximiza el número de contenedores cargados y otro algoritmo que maximiza el número de toneladas cargadas, ambos basados en la técnica *Greedy*.



Maximizar el número de contenedores cargados

- **Pseudocódigo**

```
Pedimos los datos de los pesos al usuario:
        k: capacidad del buque
        n: número de contenedores
        w[i]: cada uno de los pesos de los contenedores, 1 ≤ i ≤ n
Ordenamos los pesos de los contenedores de menor a mayor (C)
Aplicamos Greedy:
        num_contenedores = 0
        Mientras num_contenedores no sea una solución y queden
        contenedores por ver:
                num_toneladas += w[i]
                si num_toneladas ≤ k es factible entonces ++num_contenedores
        Devolvemos num_contenedores
```

- **Explicación del código**

En primer lugar, declaramos las variables que se van a usar en el código e inicializamos el ***conjunto solución*** a cero. En este caso, inicializamos la variable *num_contenedores=0*, donde al final del algoritmo tendremos el número máximo de contenedores que podremos meter en el buque sin pasarnos de su capacidad. Posteriormente pedimos los datos de los pesos al usuario, usando cout y cin.

Seguidamente, ordenamos el vector de los pesos de los contenedores de menor a mayor. Este será el ***conjunto de candidatos*** para nuestro algoritmo Greedy, que aplicamos a continuación con el bucle for. Mientras num_contenedores no sea solución y queden contenedores por ver, la ***función de selección*** irá cogiendo el siguiente de este vector, que coincide con el contenedor de menor peso que haya en ese momento y que no haya sido ya descartado.

Si este contenedor escogido cumple la ***condición de factibilidad***, que la suma total de los pesos cargados no exceda la capacidad del buque (incluido el peso del contenedor seleccionado), añadimos el contenedor escogido al conjunto solución (++num_contenedores); mientras que si no la cumple, descartamos este contenedor y nunca más volveremos a tomarlo como candidato.

Una vez salimos de este bucle for, mostramos la solución obtenida en la variable num_contenedores, que será la óptima, como veremos más adelante.

- **Eficiencia teórica**

```
// Declaración de datos
int w[MAXN];
int k,n;
int num_contenedores = 0;
int num_toneladas = 0;

// Pedimos los datos de los pesos al usuario
cout << "Capacidad contenedora: ";
cin >> k;
cout << endl << "Numero contenedores: ";
cin >> n;
cout << endl;

for(int i= 0; i < n; ++i){                                    //O(n)
        cout << "Peso del contenedor " << (i+1) << ": ";
        cin >> w[i];
        cout << endl;
}
```

```
// Ordenamos los pesos de los contenedores de menor a mayor
sort(w,w+n);                                          //O(nlogn)

// Aplicamos la técnica Greedy
for(int i = 0; (i < n); ++i){                         //O(n)
        num_toneladas += w[i];
                if(num_toneladas <= k)                //O(1)
                ++num_contenedores;
        else                                          //O(1)
                break;
}

// Mostramos la solución óptima
cout << "Numero de contenedores cargados: "<< num_contenedores << endl;
```

La eficiencia del algoritmo sería $O(nlogn)$, resultado de aplicar la regla del máximo entre ambos bucles de eficiencia $O(n)$, dado que en el peor de los casos se ejecutarán n veces, y la función sort cuya eficiencia es la final.

- **Ejemplo de una ejecución**

  Capacidad máxima de carga: **6**

  Vector introducido:

4	**9**	**3**	**1**	**6**

  Vector ordenado:

1	**3**	**4**	**6**	**9**

La función de selección nos va dando el siguiente contenedor del vector ordenado de menor a mayor, que es el conjunto de candidatos. Al principio selecciona el primer candidato y vemos que cumple la condición de factibilidad, pues su peso no sobrepasa la capacidad máxima del buque (1≤6). Entonces, **aumentamos en uno la variable num_contenedores** y seguimos con el siguiente candidato. El contenedor que está en la posición 1 del vector también cumple la condición de factibilidad porque la suma de su peso con los ya cargados es inferior o igual a 6, luego, **num_contenedores pasa a valer 2**.

Al pasar al tercer candidato, este **ya no cumple la condición de factibilidad** porque (1+3+4)>6 y lo descartamos como candidato, obteniendo ya la solución final: podemos introducir 2 contenedores máximo en un buque de capacidad máxima 6.
Resultado de la ejecución con nuestro código:

Capacidad contenedora: 6
Numero contenedores: 5

Peso del contenedor 1: 4
Peso del contenedor 2: 9
Peso del contenedor 3: 3
Peso del contenedor 4: 1
Peso del contenedor 5: 6

Vector ordenado:
1 3 4 6 9

Número de contenedores cargados: 2

- **Demostración de su optimalidad**

Este algoritmo Greedy siempre proporciona la solución óptima, pero esto es algo que debemos demostrar matemáticamente, ya que no siempre ocurre así con este tipo de algoritmos. Demostración de la optimalidad de este algoritmo Greedy:

Para comenzar, fijemos la siguiente notación para aclarar el problema:
xi: variable por cada contenedor que puede valer *xi=1* si el contenedor *wi* está cargado o *xi=0* si no lo está, siendo *1≤i≤n*, para *n*: número de contenedores. Tenemos: $\sum_{i=1}^{n} wixi \leq K$ para $xi \in \{0,1\}$, $1 \leq i \leq n$ y K capacidad de carga.

Sea $T = \{c1, \ldots, cn\}$ el conjunto de contenedores y supongamos, sin pérdida de generalidad, que $p1 \leq p2 \leq \ldots \leq pn$. La solución que proporciona el algoritmo Greedy viene dada por $S = \{c1, \ldots, cm\}$ de modo que: $\sum_{i=1}^{m} pi \leq K$ y $\sum_{i=1}^{m+1} pi > K$

En consecuencia tenemos que: $\sum_{i=1}^{m} pi + \sum_{i=1}^{m+1} pi > K \ \forall Q \subseteq T \backslash S, \ Q \neq \emptyset$, con lo que queda demostrado que los contenedores del conjunto S se corresponden con la solución óptima, no se podrían meter más.

Maximizar el número de toneladas cargadas

- **Pseudocódigo**

```
Pedimos los datos de los pesos al usuario:
        k: capacidad del buque
        n: número de contenedores
        w[i]: cada uno de los pesos de los contenedores, 1 ≤ i ≤ n
Ordenamos los pesos de los contenedores de mayor a menor (C)
Aplicamos Greedy:
        suma_toneladas = 0
        Mientras suma_toneladas no sobrepase k y queden contenedores por
        ver:
                si (w[i]+suma_toneladas) ≤ k es factible entonces
                        suma_toneladas += w[i]
        Devolvemos num_toneladas
```

- **Explicación del código**

En primer lugar, declaramos las variables que se van a usar en el código e inicializamos el ***conjunto solución*** a cero. En este caso, inicializamos la variable *suma_toneladas=0*, donde al final del algoritmo queremos tener el número máximo de toneladas que podremos cargar en el buque sin pasarnos de su capacidad. Posteriormente pedimos los datos de los pesos al usuario, usando cout y cin.

Seguidamente, ordenamos el vector de los pesos de los contenedores de mayor a menor. Este será el ***conjunto de candidatos*** para nuestro algoritmo Greedy, que aplicamos a continuación con el bucle for. Mientras *suma_toneladas* no sea solución y queden contenedores por ver, la ***función de selección*** irá cogiendo el siguiente de este vector, que coincide con el contenedor de mayor peso que haya en ese momento y que no haya sido ya descartado.

Si este contenedor escogido cumple la ***condición de factibilidad***, que la suma total de los pesos cargados no exceda la capacidad del buque (incluido el peso del contenedor seleccionado), añadimos el contenedor escogido al conjunto solución (*suma_toneladas += w[i]*); mientras que si no la cumple, descartamos este contenedor y nunca más volveremos a tomarlo como candidato.

Una vez salimos de este bucle for, mostramos la solución obtenida en la variable num_contenedores, que podrá no ser la óptima, como veremos más adelante.

- **Eficiencia teórica**

```
// Declaración de datos e inicialización
int w[MAXN];
int c,n;
int suma_toneladas = 0;

// Pedimos los datos de los pesos al usuario
cout << "Capacidad contenedora: ";
cin >> c;
cout << endl << "Numero contenedores: ";
cin >> n;
cout << endl;


for(int i= 0; i < n; ++i){                                    //O(n)
        cout << "Peso del contenedor " << (i+1) << ": ";
        cin >> w[i];
        cout << endl;
}

// Ordenamos los pesos de los contenedores de mayor a menor
sort(w,w+n,mayor);                                            //O(nlogn)

// Aplicamos la técnica Greedy
for(int i = 0; ((i<n)&&(suma_toneladas <= c)); ++i){          //O(n)
        if (w[i]<=c){                                         //O(1)
                if (w[i]+suma_toneladas <= c)                 //O(1)
                        suma_toneladas += w[i];
        else                                                  //O(1)
                break;
        }
}

//Mostramos la solución
cout << "Número de toneladas cargadas: "<< suma_toneladas << endl;
```

De nuevo, la eficiencia del algoritmo sería $O(nlogn)$, por los mismos motivos que en el primer apartado.

- **Ejemplo de una ejecución**

  Capacidad máxima de carga: **30**

  Vector introducido:

2	**10**	**20**	**40**	**15**	**35**	**25**

  Vector ordenado:

40	**35**	**25**	**20**	**15**	**10**	**2**

Vemos cómo el 40 y el 35 no entran ya que son mayores que 30. Sí entra el 25, pero como después 25+20=45 que es mayor que la capacidad del contenedor ya termina y solo se queda con 25.

Resultado de la ejecución con nuestro código:

```
Capacidad contenedora: 30
Numero contenedores: 7

Peso del contenedor 1: 2
Peso del contenedor 2: 10
Peso del contenedor 3: 20
Peso del contenedor 4: 40
Peso del contenedor 5: 15
Peso del contenedor 6: 35
Peso del contenedor 7: 25

Vector ordenado:
 40 35 25 20 15 10 2

Número de toneladas cargadas: 25
```

- **Demostración de su NO optimalidad**

Con el ejemplo anterior podemos poner un contraejemplo de que este algoritmo Greedy no es óptimo ya que podemos coger otras combinaciones con contenedores de pesos menores que dé más próximo a 30 que 25.

40	**35**	**25**	**20**	**15**	**10**	**2**

3. Viajante de Comercio

Dado un conjunto de ciudades y las distancias entre todas ellas, se pide averiguar la **ruta más corta** que visita todas las ciudades una única vez y vuelve al punto de partida.

Una solución podría ser considerar una ciudad como la inicial y realizar $(n-1)!$ permutaciones. Calcular el coste de cada ruta y guardar la más corta. Sin embargo, esta solución está muy alejada de la más eficiente. Supondría una complejidad de $O(n!)$.

Por esa misma razón, implementaremos algoritmos de tipo *Greedy* que resuelvan el problema con mayor eficiencia. Éstos son en orden: vecino más cercano, inserción más lejana y algoritmo propio (inserción más cercana).

Vecino más cercano

- **Explicación del algoritmo**

  La manera en la que actúa la función CalcularRecorrido es la siguiente:
  - Elegimos un nodo de comienzo.
  - Buscamos el nodo que aún no está en el camino más cercano al último nodo que se añadió. Entonces se añade al camino la línea que conecta ambos nodos.
  - Cuando todos los nodos estén en el camino, se añade por último una línea que una el primer y último nodos añadidos.

  De este modo, el algoritmo no tendría en cuenta caminos que no pasen por todos los nodos.

- **Pseudocódigo**

Función CalcularRecorrido(matriz)
1. Buscar una ciudad aleatoria *ciudad_random*
2. Actualizamos element a dicha ciudad y buscamos la más cercana no visitada que será nuestro destino (dst)
3. Marcamos dst como visitada
4. Comprobamos
    a. Si todas las ciudades se han visitado, termina
    b. Si no, vuelve al paso 2

- **Eficiencia teórica**

```
public void CalcularRecorrido(int matriz[][]){

        Random rand = new Random();
        int ciudad_random = rand.nextInt(matriz[1].length);

        nNodos = matriz[1].length - 1;
        int[] ciudadesVisitadas = new int[nNodos + 1];
        ciudadesVisitadas[ciudad_random] = 1;
        pila.push(ciudad_random);
        int element, dst = 0, i;
        int min = Integer.MAX_VALUE;
        boolean minFlag = false;
        int suma = 0;
        System.out.print(ciudad_random + "\t");

        while (!pila.isEmpty()){                                    //O(n²)

            element = pila.peek();
            i = 1;
            min = Integer.MAX_VALUE;

            while (i <= nNodos){                                    //O(n)

                if (element!=i && ciudadesVisitadas[i] == 0){       //O(1)

                    if (min > matriz[element][i]){                  //O(1)

                        min = matriz[element][i];
                        dst = i;
                        minFlag = true;
                    }
                }
                i++;
            }

            if(min==matriz[element][dst]){                          //O(1)
                suma += min;
            }else{                                                  //O(1)
                if(element == dst)                                  //O(1)
                    suma += matriz[element][ciudad_random];
            }

            if (minFlag){                                           //O(1)
```

```
                    ciudadesVisitadas[dst] = 1;
                    pila.push(dst);
                    System.out.print(dst + "\t");
                    minFlag = false;
                    continue;
                }
                pila.pop();
            }

        System.out.print("Distancia recorrida: " + suma + "\n");          //O(1)
    }
```

La eficiencia teórica del algoritmo viene dada por el primer bucle while ejecutado n veces, en cuyo interior se encuentra otro bucle while ejecutado n veces en el peor de los casos. Como los bucles están anidados, la complejidad del while es $O(n^2)$. El resto de instrucciones son de complejidad $O(1)$.

Por tanto, la eficiencia final del algoritmo será $O(n^2)$.

- **Ejecución del problema**

Para la ejecución se debe pasar como parámetro de la función, la matriz de distancias entre ciudades.

Cuando se ejecute, cada vez que se visite una ciudad, ésta se mostrará por pantalla. Finalmente, se mostrará también la distancia final recorrida.

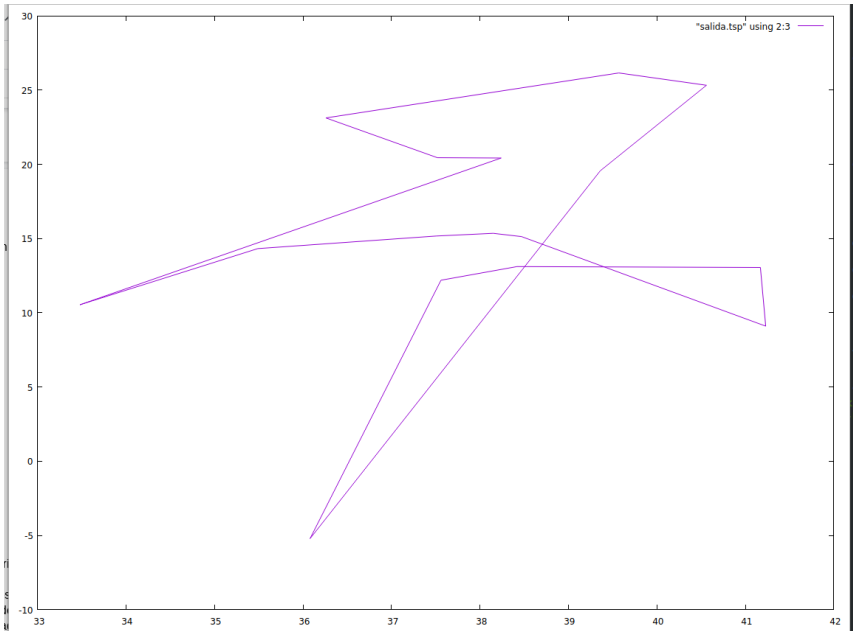
Veamos cuáles son los resultados obtenidos en cada uno de los conjuntos de datos proporcionados:

- Resultados de *ulysses16.tsp*

```
Recorrido de 16 ciudades con Vecino más cercano
6	7	10	9	12	13	14	15	5	1	8	4
2	3	16	11
Distancia recorrida: 81
```

Cabe destacar que la ciudad de partida es aleatoria, así que la distancia recorrida varía en cada ejecución. Por ello, hemos calculado que la media de la distancia es de 80.75 unidades.

Para el caso mostrado, hemos visto cómo se vería el recorrido en el plano.



Como se puede comprobar en la imagen, el recorrido está muy lejos de ser el más rápido. Esto es porque *Greedy* no siempre proporciona una solución óptima, ya que lo que busca es realizar una solución óptima a un problema complejo con pocos pasos.

Este es un claro ejemplo de una solución *greedy* con mayor eficiencia que la de fuerza bruta, que sería $O(n!)$, mientras que ésta es $O(n^2)$. Y además, es fácil de implementar. Sin embargo, tiene un riesgo alto de no obtener la ruta óptima.

Inserción más lejana

- **Explicación del algoritmo**

  El código indicado en el apartado de “Eficiencia teórica” realiza lo siguiente.

  En primer lugar, comprueba que las tres ciudades este, oeste y norte son diferentes y las añade al vector de *visitado*. En caso de que alguna coincida, ésta no se añadirá al vector, porque si no, se añadiría dos veces.

  Posteriormente, se calcula la suma del recorrido de las tres primeras ciudades y se guarda en la variable *suma*.

  Se realiza un bucle for para todas las ciudades que no estén visitadas. En cada ciudad comprueba el primer hueco en la que podría ser colocada y guarda el índice (0) y calcula el recorrido en dicho caso.

  Realiza un bucle que calcule el recorrido si se insertase la ciudad en cada uno del resto de huecos y va guardando el índice mínimo y la mínima distancia.

  Finalmente, calcula el recorrido si la ciudad se insertase entre la ciudad de partida y la última ciudad visitada.

  Se va actualizando el vector suma y se van insertando las ciudades en el lugar donde afecten menos a la distancia recorrida.

- **Estrategia**

  1. El recorrido parcial inicial se construye a partir de las tres ciudades más al este, al oeste y al norte.
  2. El siguiente nodo a insertar es el de la primera fila que no ha sido visitada.
  3. El nodo se inserta en la posición en la que provoque menor incremento de la distancia total recorrida.
     a. Para hacerlo se calcula cada posible posición y se guarda la que es mínima y su índice.

- **Eficiencia teórica**

```
void CalcularRecorrido(int matrix[][], int dim){
    int[] recorrido=new int [dim];
    int[] visitado=new int [dim];

    for(int i=0; i < dim; i++)                    //O(n)
        recorrido[i]=-1;
    for(int i=0; i < dim; i++)                    //O(n)
        visitado[i]=0;

    int contr=0;
    int contv=0;
```

```
recorrido[contr]=east_city;
contr++;
visitado[east_city]=1;
contv++;
if(west_city != -1){                                    //O(1)
        recorrido[contr]=west_city;
        contr++;
        visitado[west_city]=1;
        contv++;
}
if(north_city != -1){                                   //O(1)
        recorrido[contr]=north_city;
        contr++;
        visitado[north_city]=1;
        contv++;
}

for(int i =0; i < contr-1; i++){                        //O(n)
        suma+=matrix[recorrido[i]][recorrido[i+1]];
}
        suma+=matrix[recorrido[0]][recorrido[contr-1]];

#insertamos todas las ciudades que no estén visitadas
for(int i=0; i<dim; i++){                               //O(n²)
        if(visitado[i]!=1){                             //O(1)
                int index_min = 0;
                int min_suma=suma;
                int suma_aux=0;
                #primero guardamos la primera posible posición
                int ps = matrix[recorrido[0]][i] + matrix[i][recorrido[1]];
                min_suma = min_suma - matrix[recorrido[0]][recorrido[1]]
+ps;
                index_min=0;

                #miramos el resto de posiciones posibles excepto la
última
                for(int k=1; k < contr-1; k++){         //O(n)
                        ps = matrix[recorrido[k]][i] +
        matrix[i][recorrido[k+1]];
                        suma_aux = suma -
        matrix[recorrido[k]][recorrido[k+1]] +ps;
                        if(suma_aux < min_suma){
                                index_min = k;
                                min_suma = suma_aux;
```

```
                    }
                }

                #comprobamos la última posible posición
                ps=matrix[recorrido[0]][i] + matrix[i][recorrido[contr-1]];
                suma_aux=suma - matrix[recorrido[0]][recorrido[contr-1]]+ps;
                if(suma_aux < min_suma){                    //O(1)
                    index_min = contr-1;
                    min_suma = suma_aux;
                }

                #la menor posición se guarda
                suma=min_suma;

                #insertamos el nodo en la posición y desplazamos los que van por detrás
                for(int j = contv; j > index_min; j--){     //O(n)
                    recorrido[j]=recorrido[j-1];
                }
                recorrido[index_min+1]=i;
                contr++;
                visitado[i]=1;
                contv++;
            }
        }

        System.out.println("Longitud ruta: " + suma);

        System.out.println("Ruta: ");
        for(int i = 0; i < dim; i++)                        //O(n)
            System.out.print(recorrido[i]+1 + "\t");

        }
```

La eficiencia teórica viene dada por el cuarto bucle for ejecutado n veces, en cuyo interior se encuentran otros dos bucles for no anidados entre ellos. El mayor de ellos ejecutado n-2 veces en el peor de los casos. Por tanto, la complejidad del for será $O(n^2)$. El resto de instrucciones son bucles de $O(n)$ o instrucciones de $O(1)$.
Por tanto, la eficiencia final del algoritmo será $O(n^2)$.

- **Ejecución del problema**

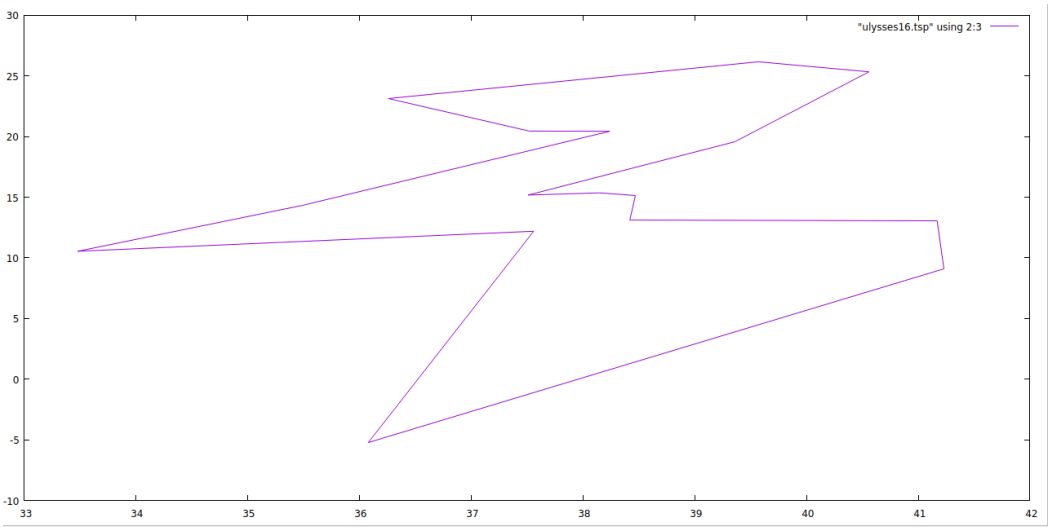
Para la ejecución se debe pasar como parámetro de la función, la matriz de distancias entre ciudades.

Cuando se ejecute, cada vez que se visite una ciudad, ésta se mostrará por pantalla. También se mostrará la distancia final recorrida.

Veamos cuáles son los resultados obtenidos en uno de los conjuntos de datos proporcionados:

❖ Resultados de *ulysses16.tsp:*

```
Recorrido de 16 ciudades con Inserción más lejana:
Longitud ruta: 69
Ruta:
9    11   6    5    15   1    8    4    2    3    16   14
13   12   7    10
```



Como podemos ver, este algoritmo está más cerca de ser óptimo y, como veremos en el apartado de comparación de las heurísticas, es más rápido. Sin embargo, es claro que su implementación es más compleja.

Algoritmo propio

- **Explicación del algoritmo**

El algoritmo que vamos a presentar a continuación es el que nosotras hemos elegido como “algoritmo propio”. Funciona de manera diferente a ‘Vecino más cercano’, ya que en vez de construir el grafo a partir de un nodo aleatorio, dado un grafo inserta en el mismo los nodos que encuentra más cercanos a él.

De esta manera, procedemos a explicar los pasos del código que hemos implementado:

- En primer lugar, escogemos cuatro nodos aleatoriamente y los unimos para formar un grafo inicial.
- Buscamos aquel nodo que esté más cerca a uno de los nodos ya insertados.
- Añadimos dicho nodo al recorrido en la posición en la que produce menor incremento de la distancia final.
- Finalmente, calculamos la distancia total teniendo en cuenta el recorrido final y la matriz de adyacencia.

- **Estrategia**

1. El recorrido parcial inicial se construye a partir de cuatro ciudades distintas buscadas de forma aleatoria.
2. El siguiente nodo a insertar es el que se acerque más a alguno de los nodos ya insertados.
3. El nodo se inserta en la posición en la que provoque menor incremento de la distancia total recorrida.
    a. Para hacerlo se calcula cada posible posición y se guarda la que es mínima y su índice.

- **Eficiencia teórica**

```
void CalcularRecorrido(int matriz[][]){

    // declaración de variables
    int num_nodos = matriz[0].length;
    int dist_total = 0;
    int contr = 0;

    int[] path = new int[num_nodos];
    boolean[] visitados = new boolean[num_nodos];

    for (int j=0; j<num_nodos;++j){                    //O(n)
        visitados[j]=false;
    }
```

```
// seleccionamos 4 nodos aleatorios y creamos el recorrido inicial
Random rand = new Random();                                    //O(1)
int random;

while(contr < 4){                                              //O(1)
        random = rand.nextInt(num_nodos-1);
        if(!visitados[random]){                                //O(1)
                path[contr] = random;
                contr++;
                visitados[random] = true;
        }
}

for(int i = 0; i < contr-1; i++){                              //O(n)
        dist_total+=matriz[path[i]][path[i+1]];
}
dist_total+=matriz[path[0]][path[contr-1]];

//en cada iteración añadimos un nodo a visitados
//realizamos este while mientras queden nodos por visitar
while (contr < num_nodos){                                     //O(n³)

        int nodo_menor = -1;
        int nodo_cercano = -1;
        int dist_menor = Integer.MAX_VALUE;

        //buscamos el hueco en el que cabe un nodo no visitado y
provoque
        //el menor aumento de distancias
        for(int i = 0; i < contr-1; i++){                      //O(n²)
                for(int j = 0; j < num_nodos; j++){            //O(n)
                        if(!visitados[j]){                     //O(1)
                                int ps=matriz[path[i]][j] +
                matriz[path[i+1]][j]-matriz[path[i]][path[i+1]];
                                if(dist_total + ps < dist_menor){  //O(1)
                                        nodo_menor = i;
                                        nodo_cercano = j;
                                        dist_menor = dist_total+ps;
                                }
                        }
                }
        }

        //para que compruebe también la dist menor teniendo en cuenta
```

```
                //el hueco entre el primer y el ultimo nodo
                for(int j = 0; j < num_nodos; j++){                    //O(n)
                        if(!visitados[j]){                             //O(1)
                                int                ps=matriz[path[0]][j]                +
                matriz[path[contr-1]][j]-matriz[path[0]][path[contr -1]];
                                if(dist_total + ps < dist_menor){
                                        nodo_menor =contr-1;
                                        nodo_cercano = j;
                                        dist_menor = dist_total+ps;
                                }
                        }
                }

                // añadimos nodo_cercano al path justo después de
        nodo_menor
                dist_total=dist_menor;

                for(int j = contr; j > nodo_menor; j--){               //O(n)
                        path[j]=path[j-1];
                }

                path[nodo_menor+1]=nodo_cercano;
                contr++;
                visitados[nodo_cercano]=true;

        }

        // mostramos por pantalla
        System.out.println("Longitud ruta: " + dist_total);            //O(1)

        System.out.println("Ruta: ");                                  //O(1)
        for(int i = 0; i < path.length; i++)                           //O(n)
                System.out.print(path[i] + "\t");
}
```

El orden del algoritmo es $O(n^3)$ dado por el bucle while principal que contiene en su interior dos bucles for anidados de orden $O(n)$ cada una. El resto de código y los demás bucles son de orden $O(n)$ u $O(1)$, que por la regla del máximo no afectarían al orden final al estar el bucle de complejidad mayor.

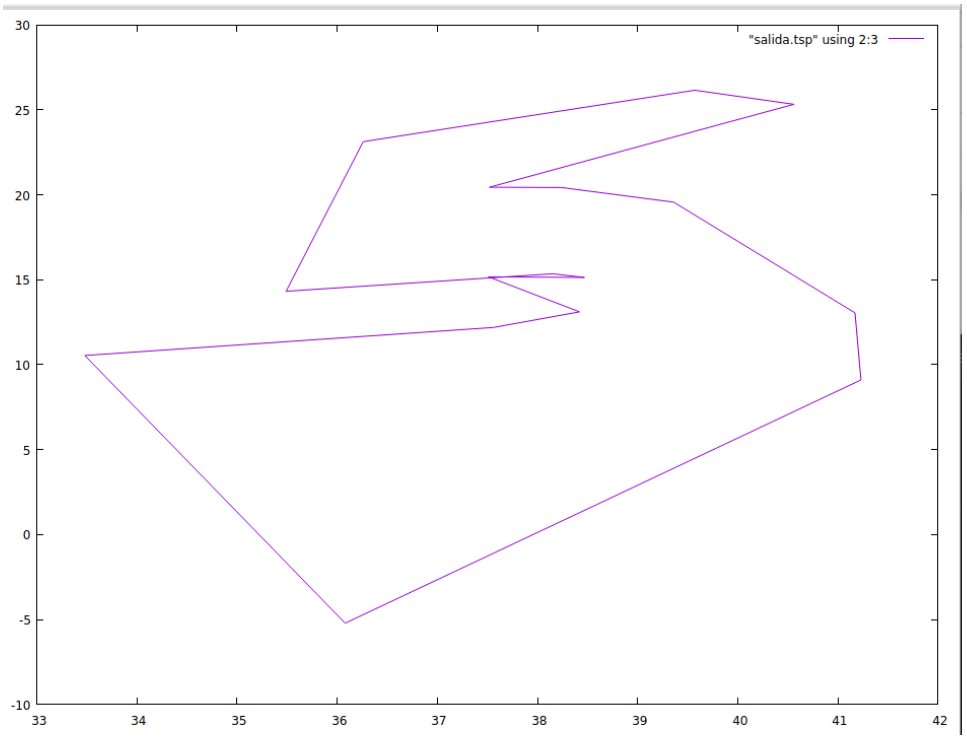
- **Ejecución del problema**

Para la ejecución pasamos como parámetro de la función la matriz de adyacencia (distancias entre ciudades).

Se muestra por pantalla la distancia final del recorrido y la ruta que sigue el mismo. Los resultadoso

❖ Resultados de *ulysses16.tsp:*

Recorrido de 16 ciudades con Algoritmo Propio
Longitud ruta: 67
Ruta:
10 16 1 8 3 2 4 15 13 12 14 7 6
5 11 9



Vemos que el algoritmo propuesto por el equipo es bastante óptimo, prácticamente igual que el otro algoritmo de inserción estudiado. Sin embargo, veremos en el siguiente apartado que el tiempo de ejecución es mayor en la mayoría de los casos. Y, además, como se habrá podido comprobar en el apartado de eficiencia teórica, la implementación es más compleja que en otros algoritmos.

Cabe destacar también en este algoritmo que los cuatro puntos iniciales son aleatorios, lo que provoca que la distancia varíe en cada ejecución. La media de la distancia recorrida en ocho ejecuciones es de 73.75 unidades de medida.

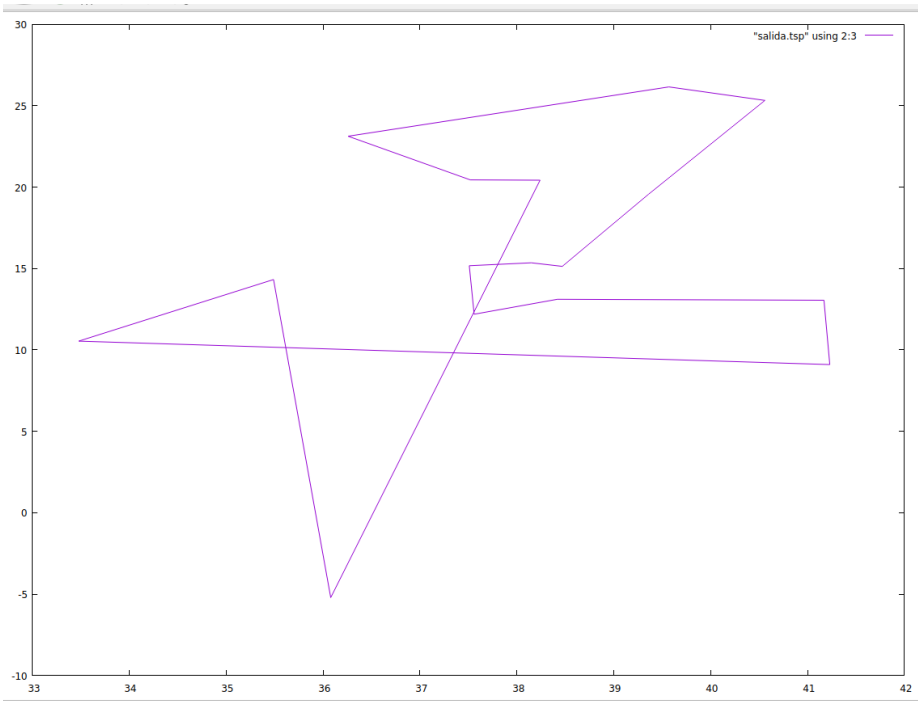
Comparación de las heurísticas descritas

Todas las soluciones han sido resultado de ejecutar los códigos en un mismo ordenador para que la comparación de tiempos tenga sentido.

Además, se han ejecutado los códigos cinco veces por fichero y algoritmo, para poder hacer una media representativa.

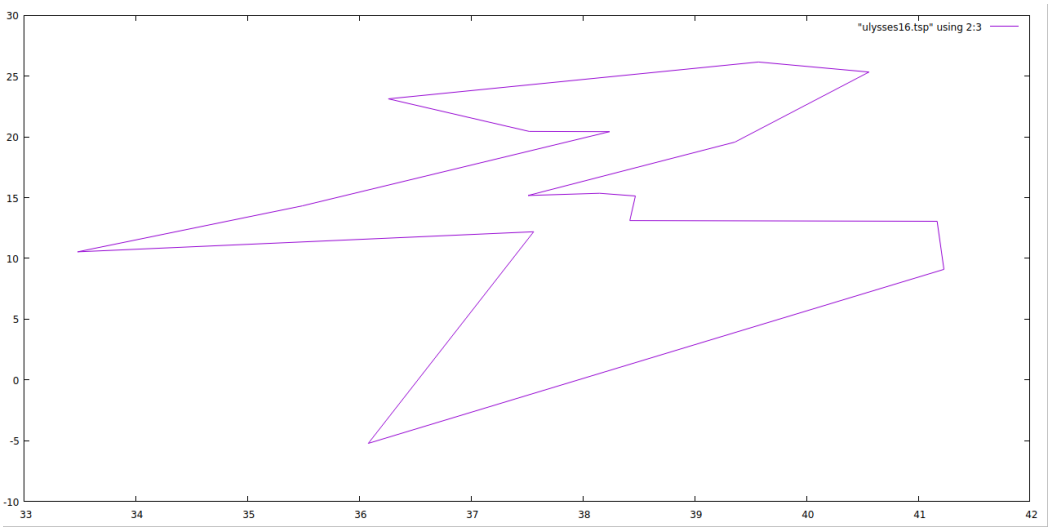
- **Resultados para** *ulysses16.tsp*
    - Vecino más cercano

Recorrido de 16 ciudades con Vecino más cercano
1 8 4 2 3 16 12 13 14 6 7
10 9 5 15 11
Distancia recorrida: 79



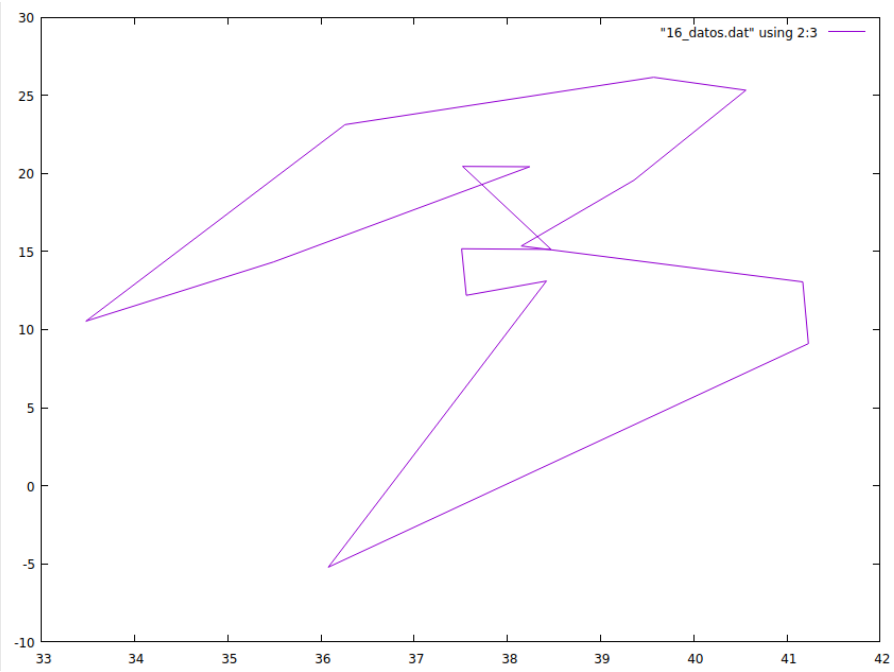
- Inserción más lejana

Recorrido de 16 ciudades con Inserción más lejana:
Longitud ruta: 69
Ruta:
9 11 6 5 15 1 8 4 2 3 16 14
13 12 7 10



- Algoritmo propio

Recorrido de 16 ciudades con Algoritmo Propio
Longitud ruta: 83
Ruta:
7 6 14 12 8 1 15 5 4 2 3 16
13 10 9 11



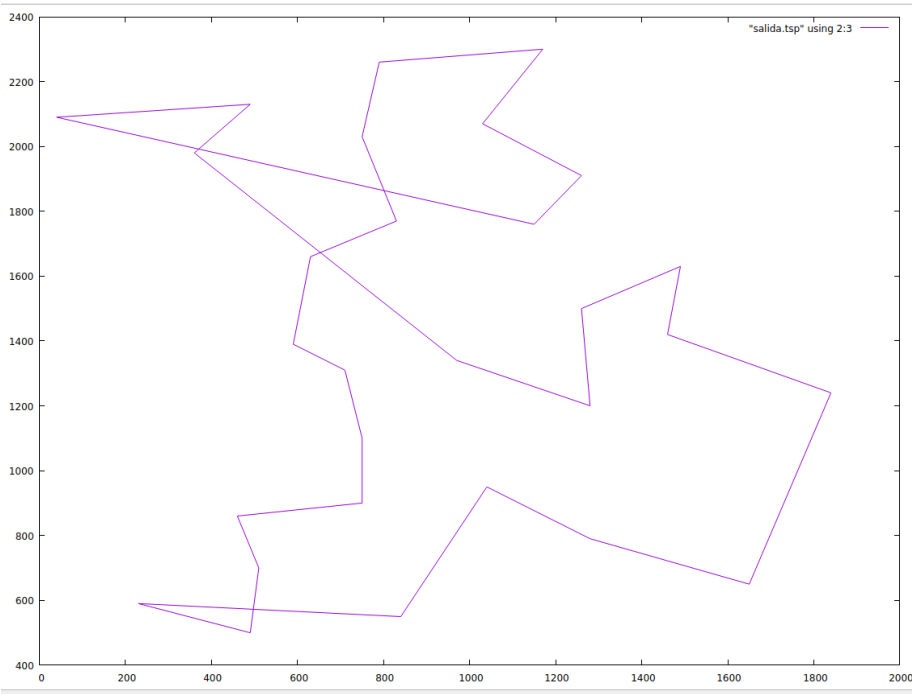
- Comparación de tiempos
  El tiempo aparece en la segunda fila con milisegundos como unidad de medida.

Vecino más cercano	Inserción más lejana	Algoritmo propio
35.826962	27.015815	33.9830832

- **Resultados para** *bayg29.tsp*

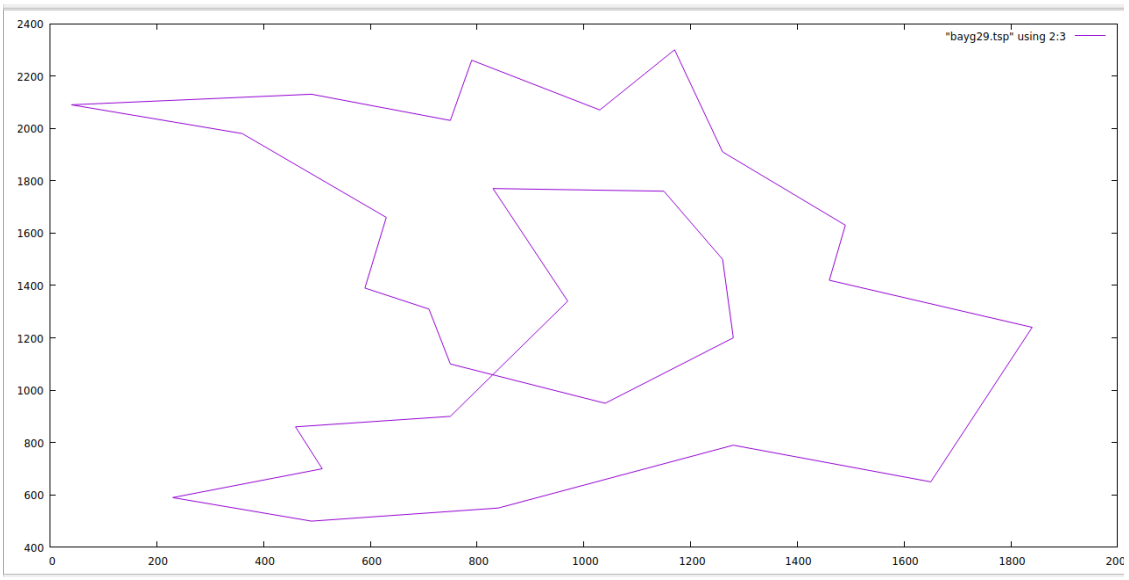
  - Vecino más cercano

Recorrido de 16 ciudades con Vecino más cercano
1 28 6 12 9 5 21 2 20 10 4 15
18 14 22 17 11 19 25 7 23 27 8 24
16 13 29 26 3
Distancia recorrida: 10200



  - Inserción más lejana

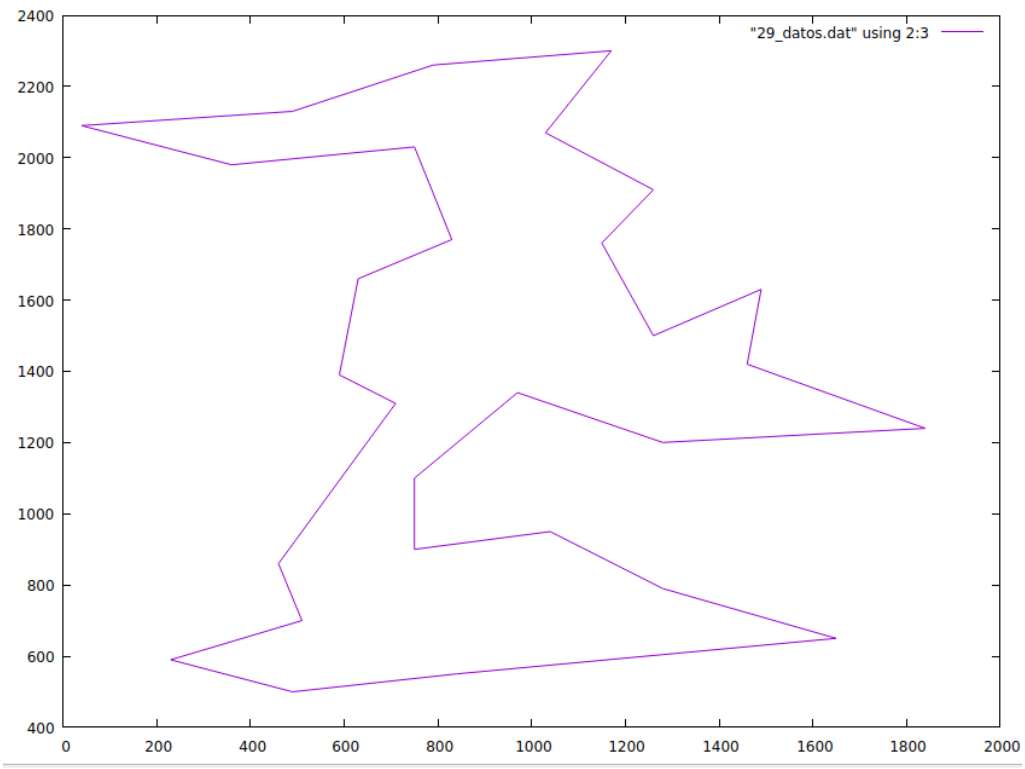
Longitud ruta: 9735
Ruta:
23 7 25 11 22 17 14 18 15 13 21 1
24 16 19 4 10 20 2 29 3 26 5 9
6 12 28 8 27



- Algoritmo propio

Longitud ruta: 9547
Ruta:

6	12	9	26	3	29	5	21	2	20	10	18
14	17	22	11	7	25	19	15	4	13	16	23
2	8	24	1	28							



- Comparación de tiempos
  El tiempo aparece en la segunda fila con milisegundos como unidad de tiempo.

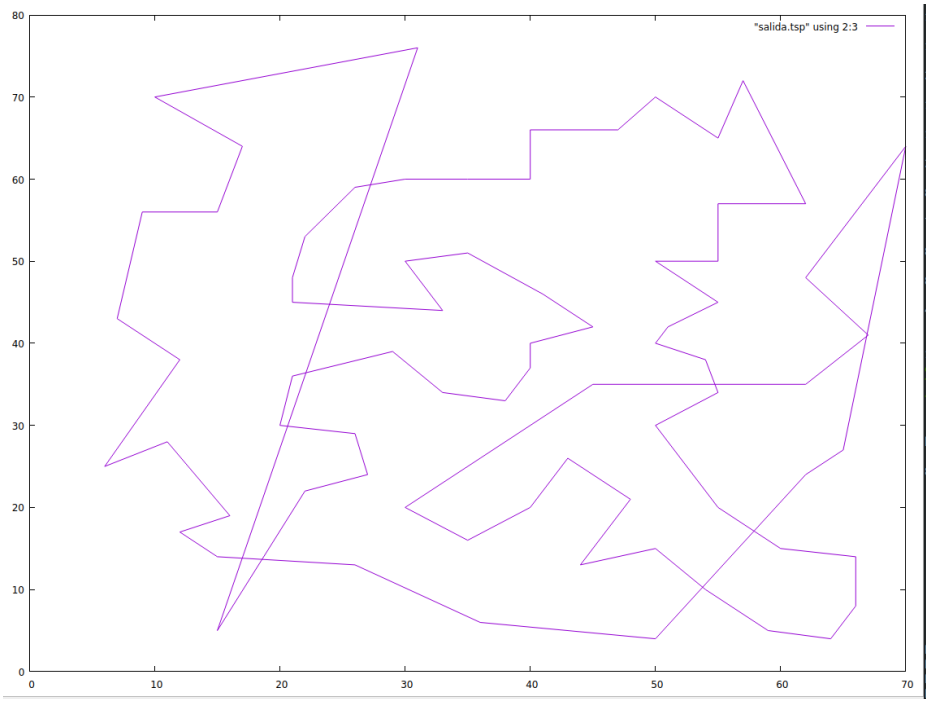
Vecino más cercano	Inserción más lejana	Algoritmo propio
36.301058	22.98003	39.7594764

- **Resultados para** *eil76.tsp*
  - Vecino más cercano

Recorrido de 16 ciudades con Vecino más cercano

1	73	33	63	16	51	6	68	75	76	67	26
12	40	17	3	44	32	9	39	72	58	10	38
65	11	66	14	53	35	7	8	46	34	52	27
45	29	5	37	20	70	60	71	36	47	21	48
30	74	28	62	2	4	13	54	19	59	57	15
69	61	22	42	41	43	23	56	49	24	18	50
25	55	31	64								

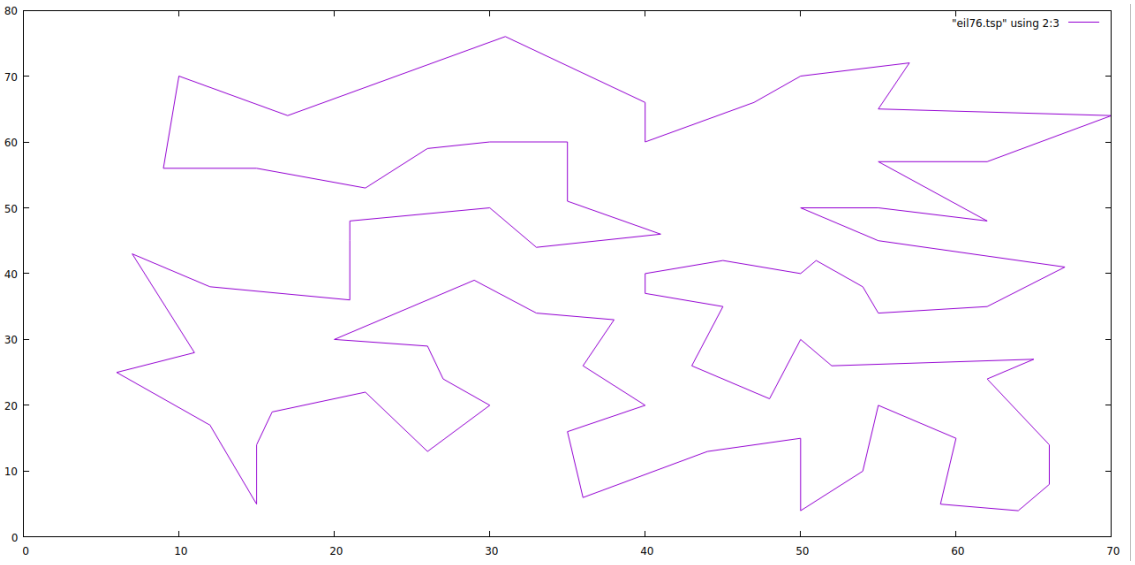
Distancia recorrida: 662



- Inserción más lejana

Longitud ruta: 581
Ruta:

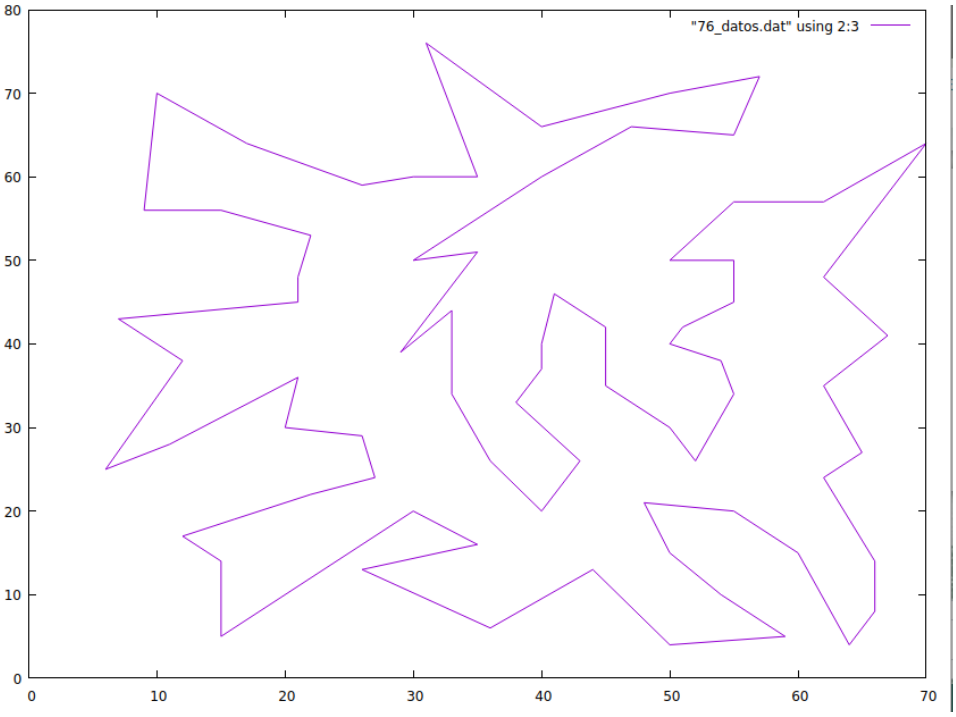
59	14	53	19	35	7	8	54	13	27	52	46
34	67	76	75	4	30	48	45	29	57	15	20
70	60	71	37	5	36	69	47	21	61	28	74
2	68	6	51	63	33	73	62	22	1	43	42
64	41	56	23	24	49	16	3	44	40	17	26
12	72	39	9	32	50	18	55	25	31	10	58
38	65	66	11								



- Algoritmo propio

Recorrido de 76 ciudades con Algoritmo Propio
Longitud ruta: 586
Ruta:

22	28	62	64	42	41	43	1	73	33	63	16
23	56	49	24	3	44	32	50	18	55	25	9
39	72	31	10	65	66	11	38	58	40	12	51
17	6	2	74	30	68	75	76	26	67	4	45
29	27	52	34	46	8	35	7	53	14	59	19
54	13	57	15	20	70	60	37	5	48	47	36
71	69	21	61								



- Comparación de tiempos
  El tiempo aparece en la segunda fila con milisegundos como unidad de tiempo.

Vecino más cercano	Inserción más lejana	Algoritmo propio
27.1894956	26.8747728	47.0789396

- **Explicación de la comparación**

En cuanto a los tiempos, hemos obtenido que el algoritmo del Vecino más Cercano es el más lento para un número de nodos pequeño (16); mientras que para un número de nodos más grande (76), este algoritmo y el de Inserción más Lejana son los más rápidos y el Algoritmo Propio, el más lento con diferencia. El motivo de estos resultados se puede justificar con la **eficiencia**, pues tanto el Vecino más Cercano como Inserción más Lejana son cuadráticos (las pequeñas diferencias que tienen se deben a las constantes ocultas) y el Algoritmo Propio es cúbico. Al principio, el cuadrático puede ser peor que el cúbico, pero para valores más grandes, el cúbico crece más rápido que el cuadrático y se agrandan las diferencias de tiempo.

En cuanto a la distancia recorrida, vemos cómo no se observa ningún patrón a simple vista. Para 16 y 79 nodos, el mejor algoritmo es el de Inserción más Lejana; y para 29, el mejor es el Algoritmo Propio. No obstante, no hay grandes diferencias de distancias entre unas heurísticas y otras, siendo las 3 buenas soluciones realmente, teniendo en cuenta que usan la técnica Greedy.

Además estos datos pueden variar ligeramente dependiendo del número de nodos y las ciudades que se elijan de forma aleatoria en los algoritmos del Vecino más Cercano y el Algoritmo Propio.

Número nodos	**Vecino más Cercano**	**Inserción más Lejana**	**Algoritmo Propio**	
16	79	69	83	*Distancia*
	35.83	27.02	33.98	*Tiempo (ms)*
29	10200	9735	9547	*Distancia*
	36.30	22.98	39.76	*Tiempo (ms)*
76	662	581	586	*Distancia*
	27.19	26.87	47.08	*Tiempo (ms)*

4. Conclusión

En primer lugar, nos hemos dado cuenta de que los **algoritmos Greedy resuelven problemas de forma muy eficiente** frente a otras posibles técnicas. Por ejemplo, como decíamos más arriba, el problema 2 podría tener eficiencia *O(n!)* si se resuelve mediante fuerza bruta, en lugar de por Greedy con eficiencia polinómica. Además, **siguen una estrategia sencilla**, implementándose siguiendo un diseño previamente establecido: conjunto candidatos, función selección, criterio de factibilidad y función objetivo.

Por otro lado, hemos podido probar que **los algoritmos Greedy no siempre dan con la solución óptima**. Por ello, ha sido necesario demostrar la corrección del algoritmo para ver cómo de cerca está la solución de la óptima o si es la óptima. Por ejemplo, en el problema de los contenedores, maximizando el número de contenedores el algoritmo Greedy sí da la opción óptima, pero maximizando las toneladas no ocurre así. De hecho, en ocasiones la solución proporcionada por estos algoritmos está lejos de ser la óptima, como nos ha ocurrido en el algoritmo del Vecino más Cercano del problema 2. No obstante, esta desventaja es una ventaja con problemas en los que es imposible (o muy difícil) alcanzar el óptimo. En estos casos, se denominan **heurísticas Greedy**.

También hemos observado que **la técnica Greedy es muy útil en problemas sobre grafos**, ya que cumplen las características de un problema Greedy perfectamente: el *conjunto de candidatos* lo conforman todos los nodos; el *conjunto solución* empieza siendo 0 y luego se va rellenando de nodos según la *función de selección*, que puede coger el nodo más cercano a los ya seleccionados, el más lejano.. según el algoritmo, y el *criterio de factibilidad*, que se asegura de que no se formen ciclos. Así se sigue hasta que todos los nodos se hayan metido en el conjunto solución.

En definitiva, l**os algoritmos que se basan en la técnica Greedy son una muy buena opción a tener en cuenta para los problemas Greedy**, pues, aunque puede que no den con la solución óptima, nos aseguran una solución eficiente y muy cercana a la óptima en la mayoría de los casos.