ugr | Universidad de Granada

ALGORÍTMICA

Curso Académico 2021/2022

Práctica 3

PRÁCTICA 3: ALGORITMOS VORACES (GREEDY)

Grupo: Buganvilla

Autores: Manuel Vicente Bolaños Quesada, Pablo Gálvez Ortigosa, Carlos García Jiménez

Índice

1. Introducción **4**

2. Problema de los contenedores **5**

2.1. Ejercicio 1 6

2.2. Ejercicio 2 7

3. Problema del viajante de comercio (TSP) **9**

3.1. Enfoque 1: vecino más cercano 10

3.2. Enfoque 2: estrategia de inserción 11

3.3. Enfoque adicional 13

3.4. Comparación 15

4. Conclusiones **26**

5. Herramientas y ordenadores utilizados **27**

5.1. Herramientas 27

5.2. Ordenadores de los integrantes del grupo 27

1. Introducción

En esta práctica trataremos de resolver dos problemas mediante la **técnica Greedy** (algoritmos voraces). Un algoritmo voraz es un **método de resolución de problemas**, que consiste básicamente en **seleccionar en cada momento lo mejor de entre un conjunto de candidatos**, sin tener en cuenta lo ya hecho, hasta obtener una solución para el problema. Este enfoque **solo se puede aplicar a problemas que reúnan un conjunto de características** determinadas. Mediante Greedy, se obtienen **soluciones cercanas a la óptima** (a veces llegan a ser la solución óptima) de una manera **bastante eficiente**.

El primero de los problemas consistirá en **montar un número de contenedores en un barco con una carga limitada**, siendo dicha carga menor que la suma de los pesos de todos los contenedores. Implementaremos dos algoritmos intentando maximizar, en un caso, el número de contenedores, y en otro la carga, y justificaremos su eficiencia en **términos de optimalidad**, demostrando rigurosamente si el algoritmo es óptimo o, en caso contrario, proporcionando un contraejemplo.

El segundo problema será el conocido como el **problema del viajante de comercio** (TSP), que consistirá en hallar el **ciclo hamiltoniano de mínimo peso** de un grafo conexo y ponderado. Utilizaremos tres **heurísticas** distintas para aproximarnos a la solución, comprobando de esta forma la gran importancia de estos procedimientos en Algorítmica. Para este segundo problema, realizaremos una somera **comparación entre las tres heurísticas**, contrastando los distintos resultados y los distintos tiempos de las tres variantes.

2. Problema de los contenedores

Supongamos que tenemos un buque mercante cuya capacidad de carga es de K toneladas, y un conjunto de contenedores $c_1, \ldots, c_n$, cuyos pesos son $p_1, \ldots, p_n$, respectivamente (expresados en toneladas). Supongamos además que la suma total de los pesos de los contenedores es mayor que la capacidad del buque. Teniendo esto en cuenta, hay que resolver dos ejercicios con algoritmos greedy:

1. Diseñar un algoritmo que maximice el número de contenedores cargados
2. Diseñar un algoritmo que intente maximizar el número de toneladas cargadas.

El paso lógico es comprobar primero si el problema reúne todas las **características de un problema greedy**:

- El **conjunto de candidatos** estará formado por los distintos contenedores.
- Los **candidatos ya usados** son los contenedores que añadimos al barco, eliminándolos del conjunto anterior, o bien los contenedores que comprobamos que, de añadirlos, superaríamos la capacidad del barco.
- En ambos casos, el **criterio de parada** será que añadiendo cualquiera de los contenedores que aún no hemos subido, se supere la capacidad de carga del barco.
- Un conjunto de contenedores es **solución** si no supera la capacidad de carga del barco.
- La **función de selección**, en el primer caso, determinará el contenedor con menor peso que aún no se ha montado, y en el segundo caso, el contenedor con mayor peso.
- La **función objetivo** asocia, a un conjunto de contenedores, en el primer caso el número de contenedores cargados y en el segundo caso, el número de toneladas cargadas.

Una vez comprobado que podemos resolver ambos problemas con un enfoque *greedy*, veamos más detalladamente cada uno de los algoritmos implementados:

2.1. Ejercicio 1

Para maximizar el número de contenedores cargados, lo más lógico es cargar los contenedores de menor a mayor peso (de hecho demostraremos que esta solución es la **óptima**). Para ello, primero ordenamos el vector que almacena los pesos, con la función *sort* de la stl. Una vez hecho esto, solo quedaría cargar los contenedores con peso mínimo, uno a uno, hasta que el peso total cargado hasta el momento sobrepase la capacidad de carga del buque, o bien hasta que hayamos cargado todos los contenedores (es este caso, hemos supuesto, al enunciar el problema anteriormente, que la suma total de los pesos de los contenedores es mayor que la capacidad del buque, así que esta última condición no se va a cumplir nunca. Sin embargo, por hacer el programa más completo, hemos hecho esa comprobación). Más explícitamente, la implementación de esto sería la siguiente:

```
int sum = 0;
bool stop = false;

sort(weights.begin(),weights.end()) ;        //O(nlogn)
vector<int>::iterator it = weights.begin();

while (it != weights.end() && !stop) {  //O(n)

    int min = *it ;

    if (sum+min <= CAPACITY) {
        sum += min ;
        total_containers++;
        ++it ;
    }
    else
        stop = true ;

}
```

Vamos a demostrar que este algoritmo consigue la solución óptima para este problema.

Sin pérdida de generalidad, podemos asumir que $p_1 \leq p_2 \leq \cdots \leq p_n$. Mediante el algoritmo anterior, conseguimos cargar los k primeros contenedores, con $k \geq 1$. Es decir

$$\sum_{i=1}^{k} p_i \leq K \qquad \text{y} \qquad \sum_{i=1}^{k+1} p_i > K$$

Sea ahora $n \geq m > k$, y vamos a demostrar que no existen m contenedores que se puedan cargar en el buque sin sobrepasar la carga. Sea por tanto $\varphi : \{1, \ldots, m\} \to \{1, \ldots, n\}$ una aplicación inyectiva y creciente, de forma que $\varphi(i) \geq i$. Esta aplicación lo que hace es ordenar los m contenedores que queremos cargar. Como la sucesión $\{p_i\}_{i=1}^{n}$ es creciente, y φ también, tenemos:

$$\sum_{i=1}^{m} p_{\varphi(i)} \geq \sum_{i=1}^{m} p_i \geq \sum_{i=1}^{k+1} p_i > K$$

luego no podemos cargar un número de contenedores mayor que k. ■

Por último, estudiemos la eficiencia teórica de este algoritmo. La función *sort* de la stl es $O(n \log n)$. Además, el bucle *while* se ejecutará, como mucho, n veces, y por tener el cuerpo de dicho bucle eficiencia $O(1)$, deducimos que la eficiencia del algoritmo es $\mathbf{O(n \log n)}$.

2.2. Ejercicio 2

En este caso, queremos maximizar el peso cargado en el buque, usaremos un algoritmo simple. Este algoritmo consiste en, primeramente, ordenar de mayor a menor el vector de pesos dado, usando las función *sort* de la stl. Finalmente, se seleccionarán los contenedores del vector, en orden, cargando en el barco los que sea posible en función del peso disponible. En esta imagen se puede observar la implementación de este algoritmo:

```
sort(weights.begin(),weights.end(),greater<int>); //O(nlog(n))

vector<int>::iterator aux_it = weights.begin();
int sum = 0;

while (aux_it != weights.end()){ //O(n)


    int we = *aux_it;

    if (sum + we <= CAPACITY){
        sum += we;
        ++aux_it;
    }
    else{
        ++aux_it;
    }

}

cout << "Peso cargado: " << sum << endl;
```

A pesar de todo, este algoritmo no es el óptimo, como podemos ver con este contraejemplo: Supongamos que con una capacidad de 14, se dispone del vector de pesos (ya ordenados) $\{7, 5, 4, 3\}$, el algoritmo daría como resultado que se cargarían $\{7, 5\}$, cuando lo óptimo sería cargar $\{7, 4, 3\}$.

Para finalizar con este algoritmo, estudiemos la eficiencia teórica. Por un lado tenemos la función *sort*, con una eficiencia de $O(n \log n)$, mientras que por otro lado tenemos el bucle *while* que se ejecutará n veces. Por tanto podemos concluir que la eficiencia del algoritmo es $\mathbf{O}(\mathbf{n} \log \mathbf{n})$

3. Problema del viajante de comercio (TSP)

El **problema del viajante de comercio** (TSP, por Traveling Salesman Problem) se define como sigue: dado un conjunto de ciudades (nodos) y una matriz con las distancias entre todas ellas (matriz de adyacencia del grafo), un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma que la distancia recorrida sea mínima. En concreto, tendremos que encontrar el **ciclo hamiltoniano de mínimo peso del grafo**.

Por ser un problema **NP-Completo**, el diseño de algoritmos exactos no es factible en problemas de cierto tamaño. Por ello, nos centraremos en **tres algoritmos de tipo greedy**, de los que evaluaremos su rendimiento. Utilizaremos tres **heurísticas** diferentes para la resolución:

1. Heurística del **vecino más cercano**
2. Heurística de **inserción** (utilizando el criterio de *inserción más económica*)
3. Heurística del **punto más cercano a los extremos del circuito**

Cada heurística se explicará con precisión en los siguientes apartados.

De nuevo, el primer paso lógico sería comprobar las **características propiamente greedy** de este problema:

- Los **candidatos** en este caso serán los nodos (ciudades) del grafo.
- Los **candidatos ya usados** serán los nodos que ya hemos recorrido, que eliminaremos de los posibles candidatos.
- El **criterio de parada** será que el tamaño del recorrido sea igual al número de nodos.
- Un conjunto de nodos podrá llegar a ser una **solución** si no repite nodos, es decir, si se visita cada nodo una única vez.
- Utilizando la matriz de adyacencia, **la función de selección** será una u otra dependiendo de la heurística utilizada (vecino más cercano, inserción más económica, ...).
- La **función objetivo** asociará a cada ciclo hamiltoniano el peso del circuito, es decir, la distancia total del recorrido.

Pasamos, por tanto, a detallar cada una de las heurísticas greedy.

3.1. Enfoque 1: vecino más cercano

En este apartado usaremos el enfoque más simple que sería en el cual el viajante visita entre los candidatos disponibles el más cercano. Es decir, tomando un nodo inicial, se usará esta función para seleccionar el nodo a menor distancia de los candidatos:

```
int min_node(int node, vector<vector<int>> & adjacency_matrix, list<int> & candidates) {

    int min_distance = INT16_MAX ;
    int min_node = -1 ;

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

        if (adjacency_matrix[node][*it] < min_distance) {
                min_distance = adjacency_matrix[node][*it] ;
                min_node = *it ;
        }
    }

    return min_node;

}
```

A continuación, se insertará el nodo seleccionado en la ruta definitiva y se eliminará de la lista de candidatos, seguidamente, se repetirá este proceso hasta que ya no queden más candidatos y finalmente, se regresará al nodo inicial al terminar la ruta.

```
/** CREACION DEL RECORRIDO INICIAL **/
list<int> tour ;
int initial_node = 0 ;


int min = INT16_MAX ;
list<int> candidates ;

for (int i=0; i<adjacency_matrix.size(); ++i)
    candidates.push_back(i) ;


tour.push_back(initial_node) ; candidates.erase(find(candidates.begin(),candidates.end(),initial_node)) ;



int aux = 0;
while (tour.size() < (adjacency_matrix.size()-1)) {
    aux = min_node(aux,adjacency_matrix,candidates);
    tour.push_back(aux); candidates.erase(find(candidates.begin(),candidates.end(),aux));
}

tour.insert(tour.end(),(*candidates.begin())) ;
tour.insert(tour.end(),(*tour.begin()));
```

Para estudiar la eficiencia teórica veamos por un lado que la función *min_node* es de complejidad $O(n)$ mientras que el *for* que contiene dicha función se ejecuta n veces, por tanto,

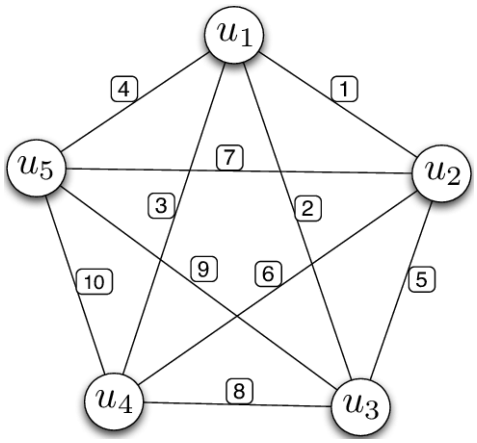
podemos concluir que la eficiencia de este algoritmo es de $\mathbf{O(n^2)}$.

3.2. Enfoque 2: estrategia de inserción

A diferencia de en el anterior enfoque, en este caso, **partiremos ya de un circuito** con 3 nodos, al que le iremos insertando nodos, hasta tener un ciclo hamiltoniano (es decir, un camino que visita todos los nodos una única vez y vuelve al nodo de origen).

El algoritmo que utilizaremos partirá de **tres nodos iniciales**: el nodo situado más al este, el más al oeste y el más al norte. Para insertar un nuevo nodo, utilizaremos el criterio de la *inserción más económica*, es decir, cada nodo lo debemos insertar en cada una de las soluciones posibles y quedarnos con el que nos permita obtener un circuito de menor longitud. Seleccionaremos el nodo que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades.

Utilizaremos un **ejemplo** para aclarar el algoritmo:



Los nodos de los que partimos serán el 1,2 y 5 por lo explicado anteriormente. La forma más económica de insertar el nodo 3 es entre el 1 y el 2 (suponiendo un incremento de 6), mientras que para insertar el nodo 4 la mejor opción es entre el 1 y el 2 con 8 de incremento. Por tanto, el algoritmo insertará el nodo 3 entre el 1 y el 2, quedando un ciclo provisional de **2,5,1,3,2**. Finalmente como solo queda un nodo se inserta el 4 (siempre supone 9 de aumento por lo que se inserta en la primera posición posible) quedando así el ciclo: **2,4,5,1,3,2**.

```
list<int> find_tour (vector<pair<double,double>> & nodes_location,
                     vector<vector<int>> & adjacency_matrix) {

    /** CREACIÓN DEL RECORRIDO INICIAL **/
    list<int> tour ;
    tour.push_back(most_east_node(nodes_location)) ;
    tour.push_back(most_west_node(nodes_location)) ;
    tour.push_back(most_north_node(nodes_location)) ;
    tour.push_back(most_east_node(nodes_location)) ;

    int num_nodes = nodes_location.size();
    list<int> candidates ;

    for (int i=0; i<num_nodes; i++)
        if (find(tour.begin(),tour.end(),i) == tour.end())
            candidates.push_back(i) ;


    /** INSERCIÓN DE NODOS **/
    while (candidates.size() > 0) {                          // O(n)
        int min_insertion = INT16_MAX ;
        int min_node = -1 ;

        pair<int,int> aux (-1,-1) ;
        pair<int,int> where_insert (-1,-1) ;

        for (auto it = candidates.begin(); it != candidates.end(); ++it) {  // O(n)
            int increment = increment_insertion(*it,tour,aux, adjacency_matrix) ; // O(n)

            if (increment < min_insertion) {
                min_insertion = increment ;
                min_node      = *it;
                where_insert  = aux ;
            }
        }

        insert (min_node, tour, aux) ;
        candidates.erase(find(candidates.begin(),candidates.end(),min_node));

    }

    return tour ;

}
```

Figura 1: Código Inserción

El código anterior simplemente realiza lo dicho: introduce los **nodos iniciales** y hace una **lista con los posibles candidatos** a entrar en el circuito (los no introducidos aún). A continuación, para cada candidato utiliza una función, llamada *increment_insertion* que calcula cuánto cuesta y donde se debería insertar el nodo de la forma más económica. Se queda con la **mínima de las inserciones** e inserta ese nodo, **borrándolo finalmente de los posibles candidatos**.

La función mencionada tiene el siguiente código:

```
int increment_insertion (int candidate,list <int> &tour, pair<int,int> &where_insert,
                         vector<vector<int>> &adjacency_matrix) {

    list<int>::iterator it2 = tour.begin() ; ++it2 ;

    int min_insertion = INT16_MAX ;
    pair<int,int> where_min (-1,-1) ;

    for (list<int>::iterator it = tour.begin(); it2!= tour.end(); ++it,++it2) {

        int price_insertion = distance(*it,candidate,adjacency_matrix) +
                              distance(candidate,*it2,adjacency_matrix) -
                              distance (*it,*it2, adjacency_matrix) ;

        if (price_insertion < min_insertion) {

            min_insertion = price_insertion ;
            where_insert.first = *it ;
            where_insert.second = *it2 ;
        }

    }

    return min_insertion ;

}
```

Figura 2: Código increment_insertion

En cuanto a la eficiencia teórica, el algoritmo tiene tres bucles anidados en los que, como mucho se recorren los n nodos, por tanto, el algoritmo es $\mathbf{O(n^3)}$. Cabe destacar que, a día de hoy, los mejores programas que implementan la inserción más económica (programas bastante más complejos que el anterior) evitan ciertas comprobaciones, y llegan a ser algo más eficientes, $O(n^2 \log n)$.

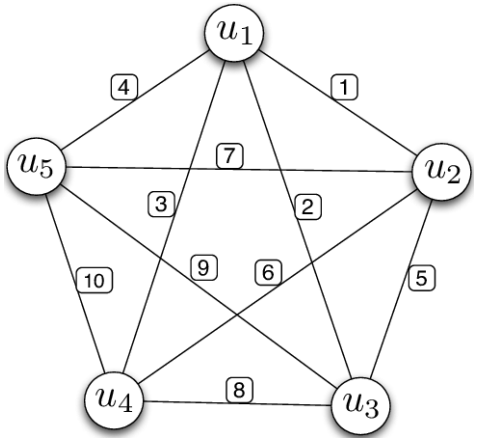
3.3. Enfoque adicional

En esta sección, nuestro objetivo será encontrar un nuevo *algoritmo greedy* distinto a los dos anteriores. Deberá ser un **algoritmo sencillo**, ya que un refinamiento excesivo podría conducir a que nuestro algoritmo pasara a ser demasiado lento (debemos ser conscientes de que **nuestro objetivo es una solución aproximada a la óptima**). Así pues, el algoritmo será similar al utilizado en el *Vecino más cercano* pero con una pequeña diferencia.

En este caso, partiremos de **dos nodos** (aunque se podía evitar para acelerar los tiempos, se ha decidido tomar la arista con menos peso del grafo). A continuación se tomará, de los posibles candidatos, es decir, los que aún no están en el circuito, **el nodo cuya distancia a uno de los**

dos nodos sea menor. Por ejemplo, si el nodo más cercano al primero del recorrido es un nodo a distancia 3 y el nodo más cercano al último nodo del recorrido está a distancia 4, se inserta en el recorrido el nodo que se encontraba a distancia 3, pasando a ser el primero del recorrido.

Utilizaremos un **ejemplo** más concreto para aclarar aún más, utilizando el siguiente grafo:



La arista con menos peso es la que une los nodos 1 y 2, por tanto, nuestro recorrido inicial será $\{1, 2\}$, el nodo más cercano al 1 es el nodo 3 (a distancia 2) y el nodo más cercano al 2 es también el 3 (a distancia 5). Por tanto, insertamos el 3 antes que el 1 y obtenemos $\{3, 1, 2\}$. Ahora, el nodo más cercano al 3 es el 4 (distancia 8) y el nodo más cercano al 2 es el 4 (distancia 6). Insertamos el nodo 4 al final, obteniendo $\{3, 1, 2, 4\}$. Por último, como solo queda un nodo se inserta al principio y al final, formando un ciclo: $\{5, 3, 1, 2, 4, 5\}$.

El código que implementa este algoritmo consistiría en lo siguiente: determinamos los dos nodos iniciales, a partir de ahí, determinamos tanto la distancia como el nodo más cercano a cada uno de los extremos (en el código lo realizará una función llamada *min_node*). Elegimos el nodo más cercano de los dos anteriores, insertándolo al principio o al final del recorrido. Cuando solo queda un nodo, insertamos el último nodo tanto al principio como al final del recorrido, cerrando así el ciclo. Una posible implementación se muestra en la siguiente página.

En cuanto a la **eficiencia teórica**, la función *min_node* es $O(n)$ y la llamada a dicha función se realiza en un bucle que realiza n iteraciones. Por tanto, el algoritmo es $\mathbf{O(n^2)}$.

```
list<int> find_tour (vector<pair<double,double>> & nodes_location,
                     vector<vector<int>> & adjacency_matrix) {

    /** CREACIÓN DEL RECORRIDO INICIAL **/
    list<int> tour ;
    int initial_node1 = -1 ;
    int initial_node2 = -1 ;

    int min = INT16_MAX ;
    list<int> candidates ;

    for (int i=0; i<adjacency_matrix.size(); ++i)
        candidates.push_back(i) ;


    for (int i=0; i<adjacency_matrix.size(); i++)
        for (int j=i+1; j<adjacency_matrix.size(); j++)
            if (adjacency_matrix[i][j] < min) {
                min = adjacency_matrix[i][j] ;
                initial_node1 = i ;
                initial_node2 = j ;
            }

    tour.push_back(initial_node1) ; candidates.erase(find(candidates.begin(),candidates.end(),initial_node1)) ;
    tour.push_back(initial_node2) ; candidates.erase(find(candidates.begin(),candidates.end(),initial_node2)) ;

    auto last_it = ++tour.begin() ;


    while (tour.size() < (adjacency_matrix.size()-1)) {

        auto min_first_node = min_node(*(tour.begin()),adjacency_matrix,candidates) ;
        auto min_last_node  = min_node(*last_it, adjacency_matrix, candidates) ;

        if (min_first_node.second <= min_last_node.second) {
            tour.insert(tour.begin(),min_first_node.first) ;
            candidates.erase(find(candidates.begin(),candidates.end(),min_first_node.first)) ;
        }
        else {
            tour.insert(next(last_it),min_last_node.first) ;
            ++last_it ;
            candidates.erase(find(candidates.begin(),candidates.end(),min_last_node.first)) ;
        }

    }

    tour.insert(tour.begin(),(*candidates.begin())) ;
    tour.insert(tour.end(),(*candidates.begin())) ;

    return tour ;

}
```

Figura 3: Código de 3ª alternativa

3.4. Comparación

Ahora realizaremos una **comparación de los tres algoritmos** utilizando tres ficheros con las **coordenadas de un cierto número de ciudades**: uno con 16 ciudades, otro con 29 ciudades y otro con 76. En los tres casos, supondremos que el grafo que conforman las ciudades es **completo**, es decir, desde una ciudad se puede viajar a cualquier otra ciudad.

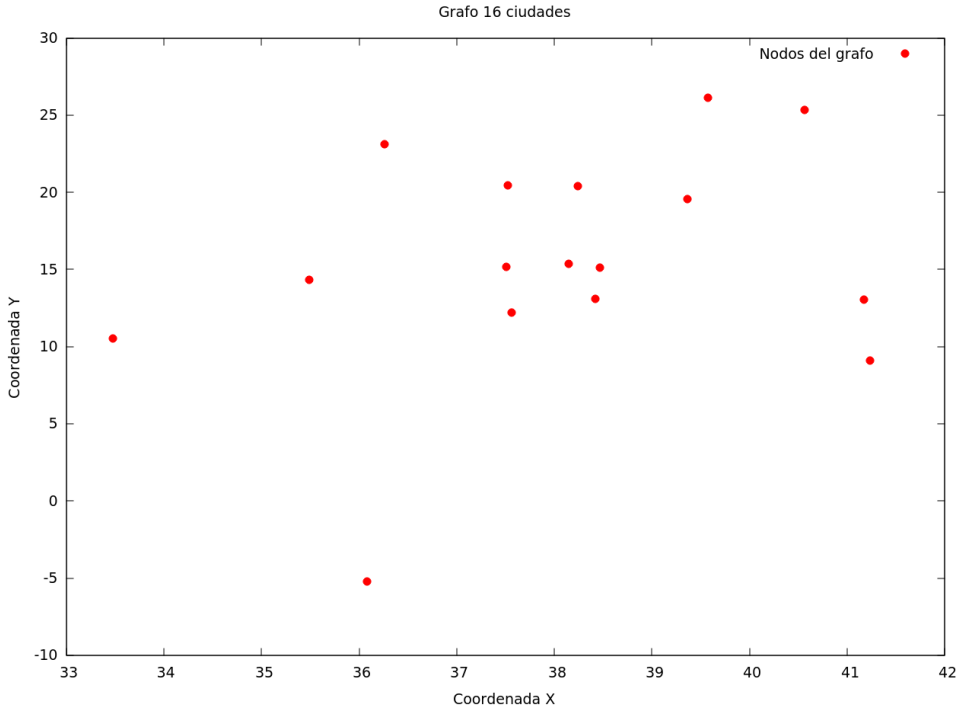
Para calcular las distancias entre las ciudades calcularemos **la parte entera de la distancia euclídea** entre los distintos nodos, obteniendo de esta forma **la matriz de adyacencia**, a la que podremos ya aplicar las tres heurísticas anteriores.

Mostraremos, para cada fichero, tanto la **nube de puntos** como las matrices de adyacencia

(o partes de ella) obtenidas:

1. **ulysses16.tsp**

   - Nube de puntos

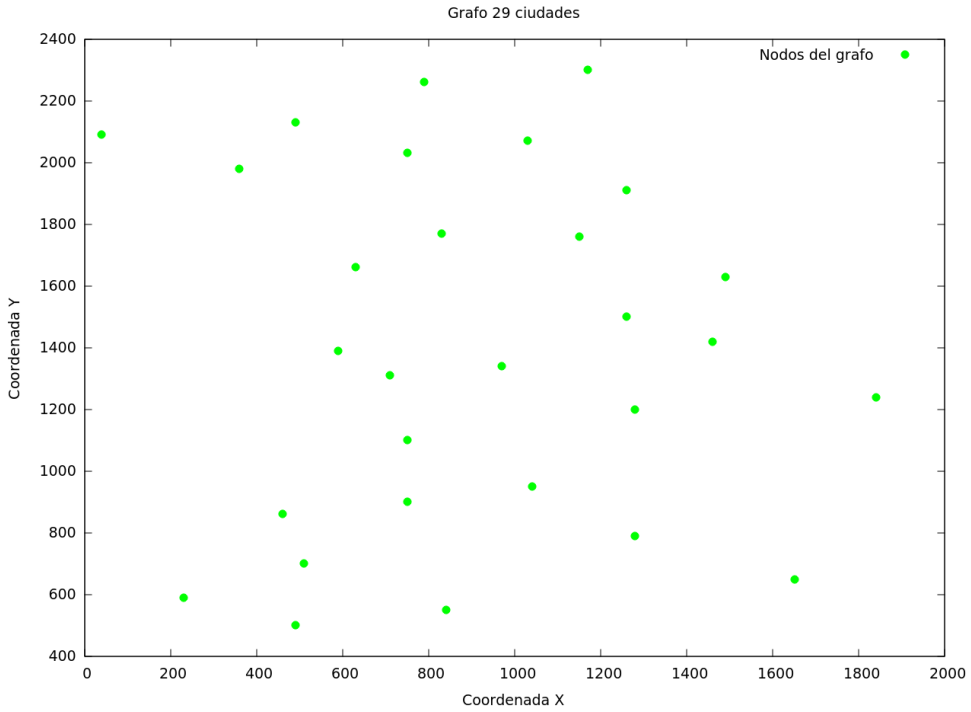


   - Matriz de adyacencia

$$\begin{pmatrix}
\infty & 5 & 5 & 3 & 10 & 8 & 7 & 0 & 11 & 7 & 25 & 5 & 5 & 5 & 6 & 1 \\
5 & \infty & 1 & 4 & 16 & 14 & 13 & 6 & 17 & 13 & 31 & 11 & 10 & 11 & 12 & 6 \\
5 & 1 & \infty & 4 & 16 & 13 & 12 & 5 & 16 & 12 & 30 & 10 & 10 & 10 & 12 & 5 \\
3 & 4 & 4 & \infty & 12 & 11 & 10 & 2 & 14 & 11 & 28 & 8 & 7 & 8 & 8 & 4 \\
10 & 16 & 16 & 12 & \infty & 4 & 5 & 10 & 7 & 8 & 15 & 6 & 6 & 6 & 4 & 10 \\
8 & 14 & 13 & 11 & 4 & \infty & 1 & 8 & 4 & 3 & 17 & 3 & 3 & 2 & 2 & 7 \\
7 & 13 & 12 & 10 & 5 & 1 & \infty & 7 & 4 & 2 & 18 & 2 & 2 & 2 & 3 & 6 \\
0 & 6 & 5 & 2 & 10 & 8 & 7 & \infty & 11 & 8 & 25 & 5 & 5 & 5 & 6 & 2 \\
11 & 17 & 16 & 14 & 7 & 4 & 4 & 11 & \infty & 3 & 15 & 6 & 6 & 7 & 7 & 10 \\
7 & 13 & 12 & 11 & 8 & 3 & 2 & 8 & 3 & \infty & 18 & 3 & 3 & 4 & 5 & 6 \\
25 & 31 & 30 & 28 & 15 & 17 & 18 & 25 & 15 & 18 & \infty & 20 & 20 & 20 & 19 & 24 \\
5 & 11 & 10 & 8 & 6 & 3 & 2 & 5 & 6 & 3 & 20 & \infty & 0 & 0 & 3 & 4 \\
5 & 10 & 10 & 7 & 6 & 3 & 2 & 5 & 6 & 3 & 20 & 0 & \infty & 0 & 2 & 4 \\
5 & 11 & 10 & 8 & 6 & 2 & 2 & 5 & 7 & 4 & 20 & 0 & 0 & \infty & 2 & 4 \\
6 & 12 & 12 & 8 & 4 & 2 & 3 & 6 & 7 & 5 & 19 & 3 & 2 & 2 & \infty & 6 \\
1 & 6 & 5 & 4 & 10 & 7 & 6 & 2 & 10 & 6 & 24 & 4 & 4 & 4 & 6 & \infty
\end{pmatrix}$$

2. **bayg29.tsp**

- Nube de puntos

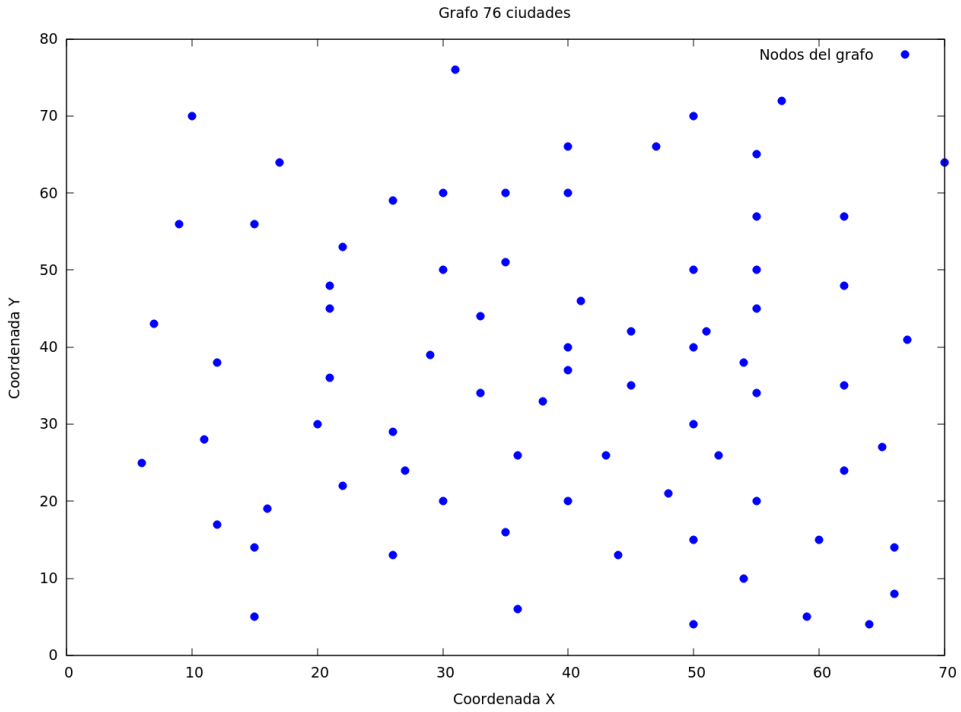


- Matriz de adyacencia

$$\begin{pmatrix}
\infty & 529 & 1158 & 771 & 482 & 332 & 1217 & 364 & 616 & 629 & \dots & 320 & 1422 & 864 & 282 & 978 & 756 & 460 & 186 & 820 \\
529 & \infty & 730 & 572 & 388 & 572 & 1435 & 860 & 620 & 359 & \dots & 228 & 1168 & 1280 & 650 & 1086 & 490 & 864 & 677 & 418 \\
1158 & 730 & \infty & 1218 & 712 & 990 & 2160 & 1521 & 769 & 1028 & \dots & 852 & 1652 & 1990 & 1355 & 1796 & 451 & 1570 & 1233 & 338 \\
771 & 572 & 1218 & \infty & 930 & 1009 & 1006 & 910 & 1160 & 213 & \dots & 674 & 653 & 1098 & 648 & 614 & 1062 & 778 & 957 & 962 \\
482 & 388 & 712 & 930 & \infty & 282 & 1647 & 841 & 233 & 721 & \dots & 272 & 1551 & 1346 & 735 & 1348 & 278 & 936 & 523 & 393 \\
332 & 572 & 990 & 1009 & 282 & \infty & 1549 & 636 & 306 & 824 & \dots & 360 & 1660 & 1159 & 614 & 1304 & 543 & 779 & 280 & 676 \\
1217 & 1435 & 2160 & 1006 & 1647 & 1549 & \infty & 992 & 1825 & 1148 & \dots & 1388 & 1169 & 619 & 935 & 395 & 1880 & 793 & 1318 & 1852 \\
364 & 860 & 1521 & 910 & 841 & 636 & 992 & \infty & 941 & 843 & \dots & 674 & 1508 & 524 & 264 & 865 & 1118 & 212 & 362 & 1182 \\
616 & 620 & 769 & 1160 & 233 & 306 & 1825 & 941 & \infty & 953 & \dots & 491 & 1785 & 1463 & 893 & 1549 & 326 & 1074 & 586 & 513 \\
629 & 359 & 1028 & 213 & 721 & 824 & 1148 & 843 & 953 & \infty & \dots & 475 & 839 & 1132 & 581 & 771 & 848 & 758 & 813 & 755 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
320 & 228 & 852 & 674 & 272 & 360 & 1388 & 674 & 491 & 475 & \dots & \infty & 1314 & 1140 & 507 & 1078 & 495 & 720 & 452 & 514 \\
1422 & 1168 & 1652 & 653 & 1551 & 1660 & 1169 & 1508 & 1785 & 839 & \dots & 1314 & \infty & 1539 & 1262 & 841 & 1630 & 1336 & 1606 & 1485 \\
864 & 1280 & 1990 & 1098 & 1346 & 1159 & 619 & 524 & 1463 & 1132 & \dots & 1140 & 1539 & \infty & 635 & 718 & 1616 & 420 & 886 & 1654 \\
282 & 650 & 1355 & 648 & 735 & 614 & 935 & 264 & 893 & 581 & \dots & 507 & 1262 & 635 & \infty & 710 & 994 & 215 & 410 & 1020 \\
978 & 1086 & 1796 & 614 & 1348 & 1304 & 395 & 865 & 1549 & 771 & \dots & 1078 & 841 & 718 & 710 & \infty & 1555 & 655 & 1120 & 1504 \\
756 & 490 & 451 & 1062 & 278 & 543 & 1880 & 1118 & 326 & 848 & \dots & 495 & 1630 & 1616 & 994 & 1555 & \infty & 1202 & 800 & 198 \\
460 & 864 & 1570 & 778 & 936 & 779 & 793 & 212 & 1074 & 758 & \dots & 720 & 1336 & 420 & 215 & 655 & 1202 & \infty & 529 & 1234 \\
186 & 677 & 1233 & 957 & 523 & 280 & 1318 & 362 & 586 & 813 & \dots & 452 & 1606 & 886 & 410 & 1120 & 800 & 529 & \infty & 902 \\
820 & 418 & 338 & 962 & 393 & 676 & 1852 & 1182 & 513 & 755 & \dots & 514 & 1485 & 1654 & 1020 & 1504 & 198 & 1234 & 902 & \infty
\end{pmatrix}$$

3. **eil76.tsp**

- Nube de puntos



- Matriz de adyacencia

$$\begin{pmatrix}
\infty & 14 & 23 & 26 & 33 & 16 & 39 & 40 & 37 & 47 & \dots & 19 & 33 & 46 & 40 & 40 & 5 & 18 & 23 & 25 \\
14 & \infty & 24 & 12 & 19 & 8 & 27 & 26 & 34 & 40 & \dots & 7 & 26 & 34 & 31 & 34 & 9 & 7 & 11 & 14 \\
23 & 24 & \infty & 26 & 42 & 16 & 29 & 34 & 14 & 28 & \dots & 20 & 50 & 58 & 55 & 20 & 21 & 31 & 20 & 19 \\
26 & 12 & 26 & \infty & 18 & 12 & 15 & 14 & 30 & 31 & \dots & 7 & 31 & 34 & 33 & 26 & 21 & 15 & 5 & 7 \\
33 & 19 & 42 & 18 & \infty & 26 & 30 & 25 & 48 & 48 & \dots & 21 & 16 & 16 & 15 & 44 & 28 & 15 & 22 & 25 \\
16 & 8 & 16 & 12 & 26 & \infty & 23 & 24 & 25 & 32 & \dots & 5 & 34 & 42 & 38 & 26 & 11 & 15 & 7 & 9 \\
39 & 27 & 29 & 15 & 30 & 23 & \infty & 7 & 25 & 18 & \dots & 20 & 46 & 44 & 45 & 18 & 34 & 31 & 16 & 14 \\
40 & 26 & 34 & 14 & 25 & 24 & 7 & \infty & 32 & 25 & \dots & 20 & 41 & 38 & 40 & 25 & 35 & 29 & 17 & 15 \\
37 & 34 & 14 & 30 & 48 & 25 & 25 & 32 & \infty & 15 & \dots & 28 & 60 & 64 & 63 & 9 & 35 & 41 & 26 & 23 \\
47 & 40 & 28 & 31 & 48 & 32 & 18 & 25 & 15 & \infty & \dots & 33 & 62 & 63 & 63 & 7 & 43 & 46 & 29 & 26 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
19 & 7 & 20 & 7 & 21 & 5 & 20 & 20 & 28 & 33 & \dots & \infty & 31 & 37 & 35 & 27 & 14 & 13 & 4 & 7 \\
33 & 26 & 50 & 31 & 16 & 34 & 46 & 41 & 60 & 62 & \dots & 31 & \infty & 16 & 9 & 57 & 30 & 18 & 34 & 37 \\
46 & 34 & 58 & 34 & 16 & 42 & 44 & 38 & 64 & 63 & \dots & 37 & 16 & \infty & 7 & 60 & 42 & 28 & 38 & 41 \\
40 & 31 & 55 & 33 & 15 & 38 & 45 & 40 & 63 & 63 & \dots & 35 & 9 & 7 & \infty & 60 & 37 & 24 & 37 & 39 \\
40 & 34 & 20 & 26 & 44 & 26 & 18 & 25 & 9 & 7 & \dots & 27 & 57 & 60 & 60 & \infty & 36 & 40 & 23 & 20 \\
5 & 9 & 21 & 21 & 28 & 11 & 34 & 35 & 35 & 43 & \dots & 14 & 30 & 42 & 37 & 36 & \infty & 13 & 18 & 20 \\
18 & 7 & 31 & 15 & 15 & 15 & 31 & 29 & 41 & 46 & \dots & 13 & 18 & 28 & 24 & 40 & 13 & \infty & 17 & 20 \\
23 & 11 & 20 & 5 & 22 & 7 & 16 & 17 & 26 & 29 & \dots & 4 & 34 & 38 & 37 & 23 & 18 & 17 & \infty & 3 \\
25 & 14 & 19 & 7 & 25 & 9 & 14 & 15 & 23 & 26 & \dots & 7 & 37 & 41 & 39 & 20 & 20 & 20 & 3 & \infty
\end{pmatrix}$$

A continuación, mostraremos los **resultados obtenidos** por las distintas heurísticas para cada fichero *(Pruebas realizadas con el Ordenador de Pablo)*. En todos los casos, los nodos se han numerado en las pruebas comenzando desde el 0, por lo que el nodo 1 del fichero es el nodo 0 de la siguiente tabla, el nodo 2 del fichero es el nodo 1 de la siguiente tabla y así sucesivamente:

1. **ulysses16.tsp**

HEURÍSTICA	**CAMINO**	**COSTE**	**TIEMPO** (μs)
VECINO MÁS CERCANO	0 , 7 , 3 , 1 , 2 , 15 , 11 , 12 , 13 , 5 , 6 , 9 , 8 , 4 , 14 , 10 , 0	79	9
INSERCIÓN	9 , 2 , 7 , 13 , 14 , 8 , 10 , 4 , 3 , 5 , 6 , 12 , 11 , 15 , 1 , 0 , 9	109	12
3ª ALTERNATIVA	10 , 14 , 4 , 8 , 9 , 6 , 5 , 13 , 12 , 11 , 15 , 0 , 7 , 3 , 1 , 2 , 10	80	47

2. **bayg29.tsp**

HEURÍSTICA	**CAMINO**	**COSTE**	**TIEMPO** (μs)
VECINO MÁS CERCANO	0 , 27 , 5 , 11 , 8 , 4 , 20 , 1 , 19 , 9 , 3 , 14 , 17 , 13 , 21 , 16 , 10 , 18 , 24 , 6 , 22 , 26 , 7 , 23 , 15 , 12 , 28 , 25 , 2 , 0	10200	20
INSERCIÓN	22 , 24 , 18 , 19 , 6 , 10 , 21 , 16 , 17 , 13 , 14 , 3 , 9 , 15 , 12 , 1 , 23 , 26 , 4 , 8 , 2 , 28 , 0 20 , 11 , 25 , 5 , 27 , 7 , 22	14322	235
3ª ALTERNATIVA	2 , 25 , 28 , 10 , 16 , 21 , 13 , 17 , 14 , 3 , 9 , 19 , 1 , 20 , 4 , 8 , 5 , 11 , 27 , 0 , 23 , 26 , 7 , 15 , 12 , 18 , 24 , 6 , 22 , 2	11640	30

3. **eil76.tsp**

HEURÍSTICA	**CAMINO**	**COSTE**	**TIEMPO** (μs)
VECINO MÁS CERCANO	0 , 72 , 32 , 62 , 15 , 50 , 5 , 67 , 74 , 75 , 66 , 25 , 11 , 39 , 16 , 2 , 43 , 31 , 8 , 38 , 71 , 57 , 9 , 37 , 64 , 10 , 65 , 13 , 52 , 34 , 6 , 7 , 45 , 33 , 51 , 26 , 44 , 28 , 4 , 36 , 19 , 69 , 59 , 70 , 35 , 46 , 20 , 47 , 29 , 73 , 27 , 61 , 1 , 3 , 12 , 53 , 18 , 58 , 56 , 14 , 68 , 60 , 21 , 41 , 40 , 42 , 22 , 55 , 48 , 23 , 17 , 49 , 24 , 54 , 30 , 63 , 0	662	92
INSERCIÓN	58 , 64 , 8 , 38 , 10 , 57 , 11 , 37 , 65 , 25 , 6 , 52 , 75 , 66 , 74 , 67 , 34 , 45 , 33 , 3 , 7 , 51 , 44 , 26 , 29 , 28 , 4 , 13 , 18 , 12 , 53 , 14 , 56 , 36 , 19 , 69 , 70 , 68 , 60 , 59 , 49 , 42 , 54 , 63 , 41 , 24 , 40 , 17 , 21 , 35 , 46 , 20 , 27 , 61 , 0 , 47 , 73 , 1 , 72 , 62 , 32 , 23 , 5 , 50 , 16 , 15 , 39 , 48 , 43 , 55 , 22 , 2 , 31 , 71 , 9 , 30 , 58	1292	4414
3ª ALTERNATIVA	54 , 68 , 60 , 21 , 63 , 41 , 40 , 42 , 0 , 72 , 61 , 27 , 73 , 29 , 47 , 20 , 46 , 35 , 70 , 59 , 69 , 19 , 36 , 4 , 28 , 44 , 26 , 51 , 33 , 45 , 7 , 34 , 6 , 52 , 13 , 18 , 53 , 12 , 56 , 14 , 3 , 74 , 75 , 66 , 25 , 11 , 39 , 16 , 50 , 5 , 67 , 1 , 32 , 62 , 15 , 2 , 43 , 31 , 8 , 38 , 71 , 57 , 9 , 37 , 64 , 10 , 65 , 58 , 30 , 24 , 49 , 17 , 23 , 48 , 22 , 55 , 54	660	152

Podemos concluir que el mejor algoritmo en cuanto a **tiempos** es, como se podía suponer, el más simple de todos, el algoritmo del **vecino más cercano**. En los tres ficheros, obtiene mejores tiempos que el resto.

El algoritmo de **inserción es el más complejo de todos** y los tiempos que se obtienen aumentan en gran medida al aumentar el número de nodos, como ya se podía adelantar del apartado de *Eficiencia teórica*, por la **enorme cantidad de comprobaciones** que tiene que realizar.

En cuanto al **coste del circuito**, tanto en el fichero de 16 como en el de 29 se obtienen **mejores resultados con el vecino más cercano**. Sin embargo, en el fichero con más ciudades, la **alternativa propuesta obtiene un circuito ligeramente mejor** que el algoritmo anterior. Aunque podríamos pensar que a medida que aumentara el número de nodos el tercer algoritmo sería algo mejor, no podemos concluirlo empíricamente por la enorme limitación de las pruebas realizadas (solo tres ficheros).

En general, en las pruebas realizadas, el algoritmo del vecino más cercano es el que mejores rendimientos aporta, tanto en tiempos como en resultados.

Resaltar también que **no se realizará un estudio ni de la eficiencia empírica ni de la híbrida**, ya que dicho estudio no se puede realizar con el número tan pequeño de datos que tenemos.

Finalmente, para cada fichero y algoritmo, **representaremos los caminos obtenidos** utilizando gnuplot:

- **ulysses16.tsp**

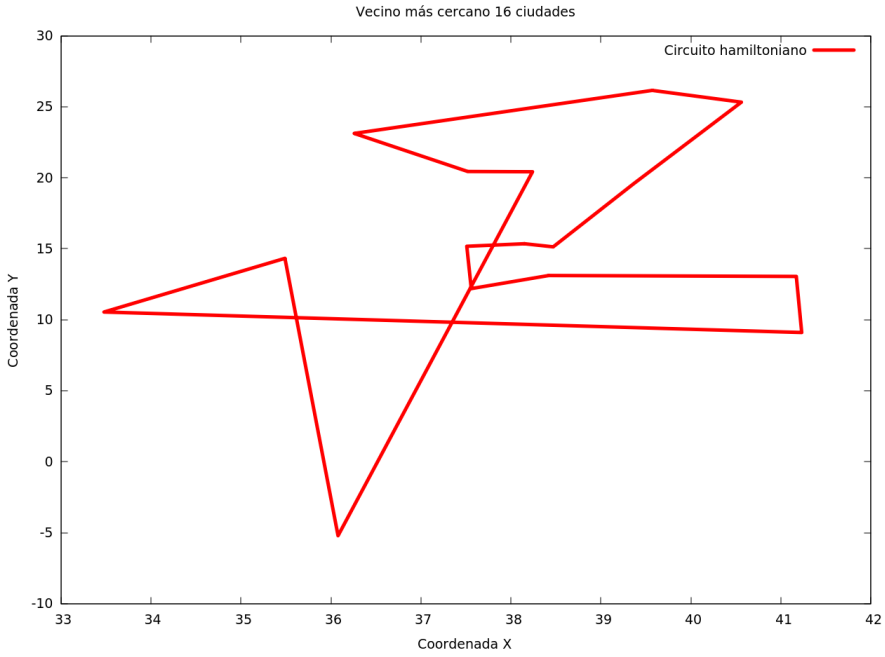


Figura 4: Camino obtenido con Vecino más cercano

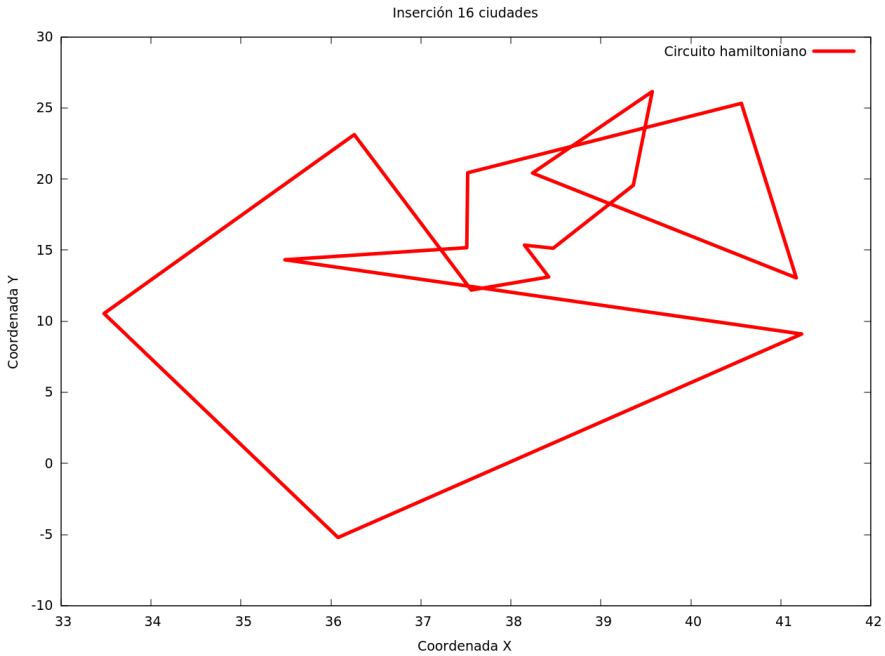


Figura 5: Camino obtenido con Inserción

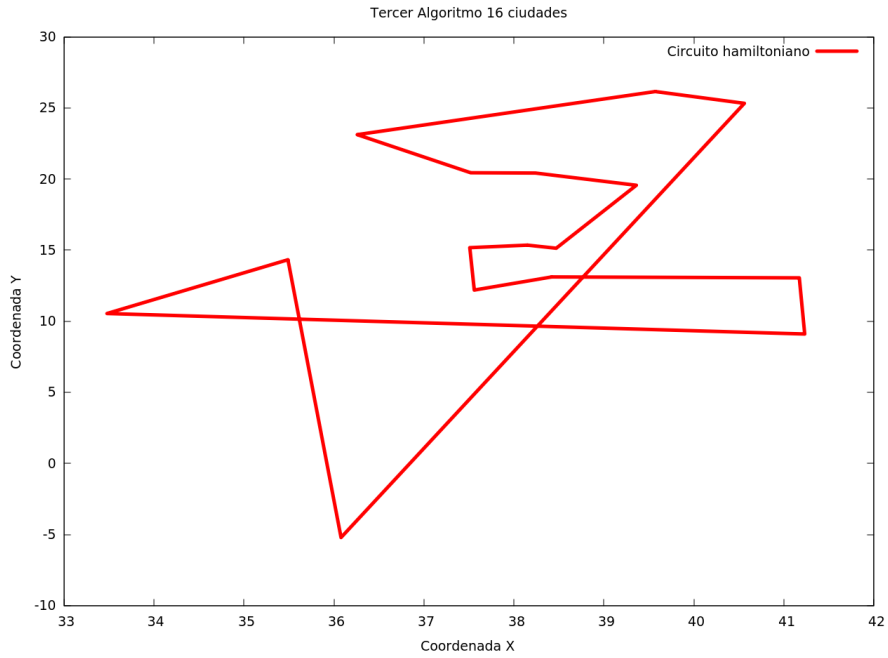


Figura 6: Camino obtenido con algoritmo propuesto

- **bayg29.tsp**

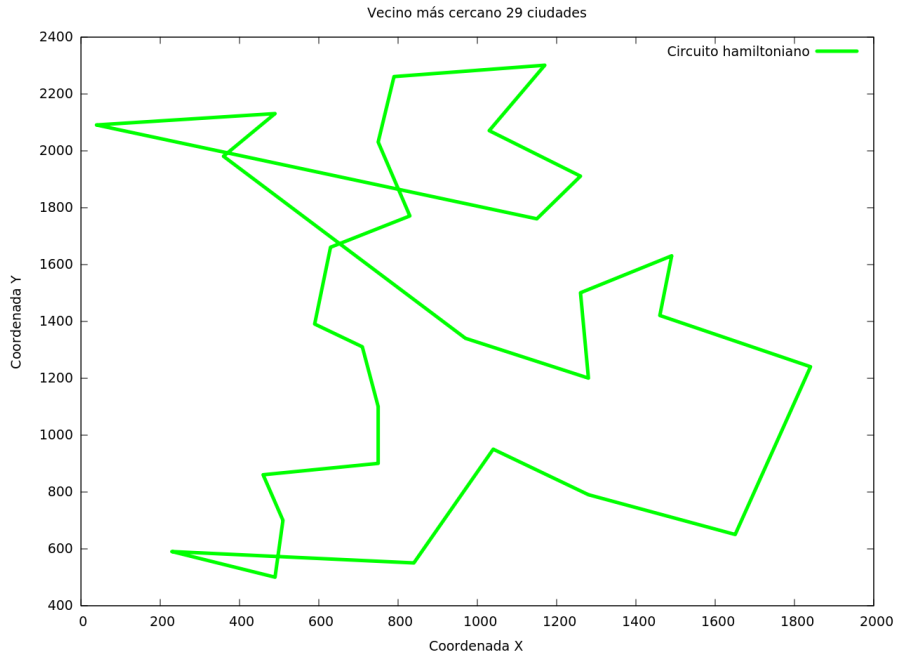


Figura 7: Camino obtenido con Vecino más cercano

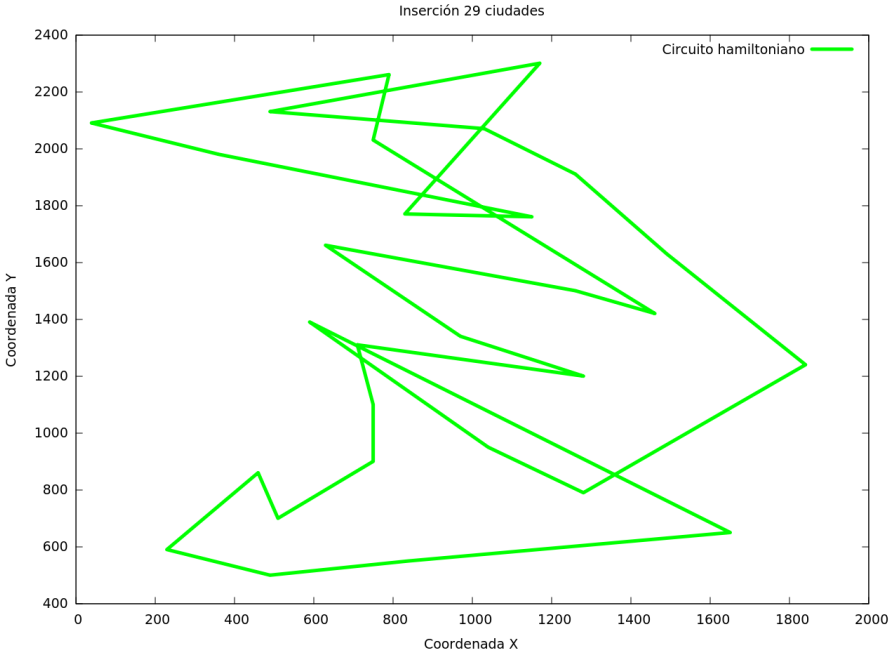


Figura 8: Camino obtenido con Inserción

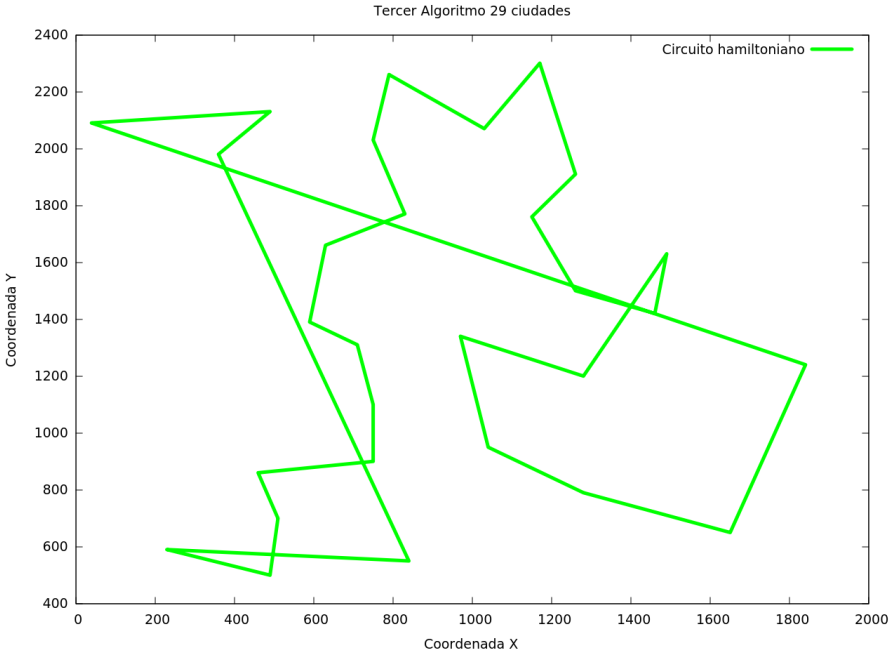


Figura 9: Camino obtenido con algoritmo propuesto

- **eil76.tsp**

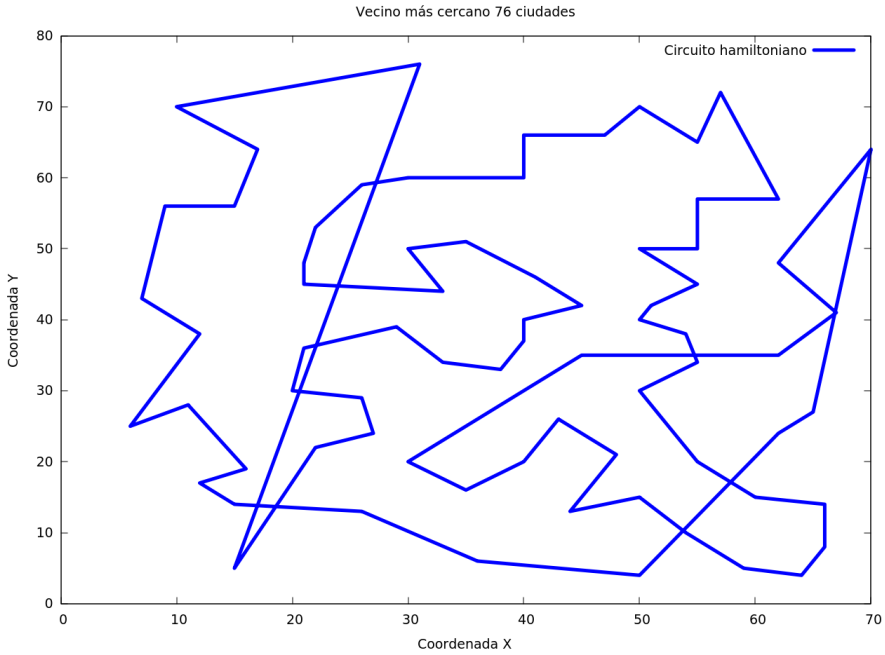


Figura 10: Camino obtenido con Vecino más cercano

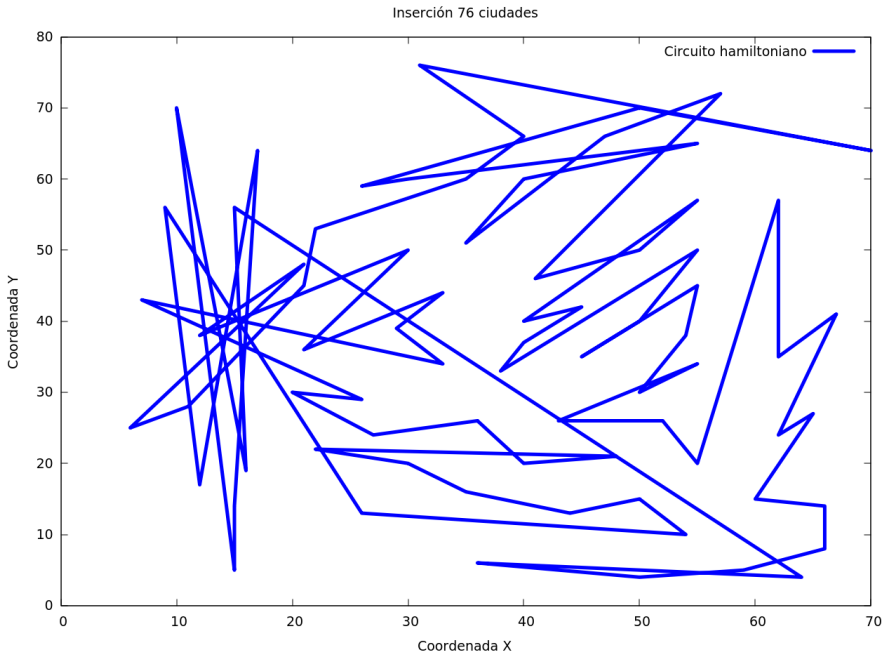


Figura 11: Camino obtenido con Inserción

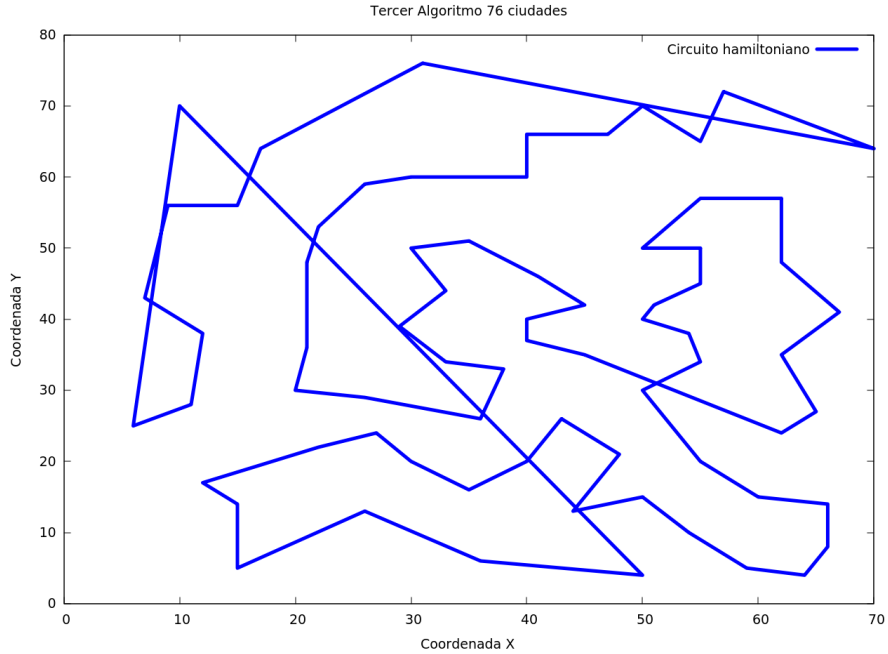


Figura 12: Camino obtenido con algoritmo propuesto

El **primer y el tercer algoritmo son bastante similares**, y fruto de ello, en los tres casos, los ciclos hamiltonianos que devuelven como solución son bastante parecidos entre ellos, como se puede apreciar en las figuras anteriores. En cuanto al algoritmo de **inserción, es en los tres casos, el grafo más distinto de los demás**, debido a que, para formar el circuito, utiliza técnicas que son bastante distintas a la de los otros dos algoritmos.

4. Conclusiones

En cada ejercicio, hemos aprendido a **verificar si un problema concreto tiene las seis características principales de un problema Greedy** (candidatos, candidatos usados, criterio solución, ...)

El enfoque "Greedy" ha probado ser muy útil para **resolver problemas de forma muy eficiente**. En el primer problema, de hecho, al intentar maximizar el número de contenedores cargados, hemos comprobado que nuestro **algoritmo es óptimo** y en el segundo caso, aunque como hemos visto, el algoritmo no es óptimo, la **solución será cercana a la óptima**.

En el segundo ejercicio, hemos podido comprobar la enorme importancia del enfoque greedy en **problemas que se plantean sobre grafos**. La solución exacta de un problema NP-completo, como es el problema del viajante de comercio, es inabarcable a día de hoy a partir de cierto número de ciudades. Sin embargo, **las tres heurísticas utilizadas nos han permitido obtener soluciones aproximadas**, cercanas a la óptima, en tiempos muy reducidos. Hemos, por tanto, comprobado la **gran utilidad de estos procedimientos** en Algorítmica.

La comparación que hemos tenido que realizar entre las tres heurísticas nos ha permitido **trabajar con las matrices de adyacencia**, que es realmente el mecanismo que utiliza el ordenador para trabajar con grafos. Además hemos podido realizar **representaciones gráficas de grafos**, algo con lo que no habíamos trabajado hasta ahora.

Greedy es, por tanto, una **técnica muy importante en la resolución de distintos problemas**. Sin embargo, siempre debemos conocer qué características debe tener un problema para poder aplicar este enfoque, si el algoritmo resultante es óptimo o no lo es, así como realizar distintas comparaciones, para decantarnos por una heurística o por otra.

5. Herramientas y ordenadores utilizados

5.1. Herramientas

- Google Spreadsheets para la organización de datos
- gnuplot 5.2 para la creación de gráficas
- Visual Studio Code para la creación y ejecución del código
- Tres ficheros de prueba del TSP suministrados por el profesorado de Algorítmica
- Overleaf (LaTeX) para la escritura de la memoria y presentación

5.2. Ordenadores de los integrantes del grupo

	MANUEL	PABLO	CARLOS
Arquitectura:	x86_64	x86_64	x86_64
Orden bytes:	Little Endian	Little Endian	Little Endian
Address sizes:	39 bits physical, 48 bits virtual	39 bits physical, 48 bits virtual	39 bits physical, 48 bits virtual
CPU(s):	8	12	12
Hilo(s) por núcleo:	2	2	2
Núcleo(s) por «socket»:	4	6	6
«Socket(s)»	1	1	1
ID de fabricante:	GenuineIntel	GenuineIntel	GenuineIntel
Modelo:	158	165	165
Nombre del modelo:	Intel i7-7700HQ @ 2.80GHz	Intel i7-10750H @ 2.60GHz	Intel i7-10750H @ 1.8GHz
Revisión:	9	2	2
CPU MHz:	2.800.000	2.600	1.800
CPU MHz máx.:	3800	5000	2600
CPU MHz mín.:	800	800	800.110
BogoMIPS:	5599.85	5199.98	5199.98
Caché L1d:	128 KiB	192 KiB	192 KiB
Caché L1i:	128 KiB	192 KiB	192 KiB
Caché L2:	1 MiB	1.5 MiB	1.5 MiB
Caché L3:	6 MiB	12 MiB	12 MiB
Memoria RAM:	16384 MB	16384 MB	8192 MB