Algoritmos Voraces (Greedy)

GRUPO: HIBISCO
Integrantes:

- Noura Lachhab Bouhmadi
- Quintín Mesa Romero
- Eduardo Rodríguez Cao
- Ramón Liria Sanchez

ÍNDICE

EJERCICIO I

“Contenedores”

ALGORITMO QUE MAXIMIZA EL NÚMERO DE CONTENEDORES

Ordenamos primero el vector con los contenedores y después vamos quedándonos con los más ligeros en el vector de salida hasta que se sobrepase la capacidad del buque.

Eficiencia teórica:
El algoritmo de Greedy es O(n) al recorrer el contenedor, pero al tener que los pesos de los contenedores de mayor a menor peso cambia la eficiencia ya que la eficiencia de la función sort es O(n.logn).

```
/**
 * @brief Opción 1: Maximizar el número de contenedores
 * @param k entero con la capacidad de carga del buque
 * @param containers vector de double con los pesos de los contenedores
 * @param containersResult vector de double con el contenedor resultante
 */

void maxNumContainers(int k, vector<double> & containers, vector<double> & conta
inersResult) {
  // Ordenación del vector dado con complejidad O(n*logn)
  sort(containers.begin(), containers.end());

  // Capacidad de carga usada
  double capacityUsed = 0;

  // Recorremos los contenedores de menor a mayor
  for (int i=0; i<containers.size() && capacityUsed+containers[i]<k; ++i){
    // Añadimos el siguiente contenedor con menor peso
    containersResult.push_back(containers[i]);

    // Incrementamos la carga usada
    capacityUsed += containers[i];
  }
}
```

ENFOQUE GREEDY

01	Conjunto de candidatos	• Contenedores disponibles
02	Conjunto de candidatos ya usados	• Contenedores que ya hemos metido en el barco
03	Función solución	• Cuando se sobrepase la capacidad de carga del buque
04	Criterio de factibilidad	• Mientras no se sobrepase la capacidad de carga
05	Función de selección	• El siguiente candidato es el contenedor más ligero
06	Función objetivo	• Maximizar el número de contenedores cargados

EJEMPLO DE EJECUCIÓN

```
eduardo@eduardo-Nitro-AN515-51:~/Escritorio/Doble_grado/Segundo_seme
Introduzca la capacidad de carga: 100
Introduzca los pesos de los contenedores: 50 32 10 10001 10 2 3 4 8
Contenedores resultantes: 2 3 4 8 10 10 32
```

ESTUDIO DE LA OPTIMALIDAD

El algoritmo de Greedy para este problema es óptimo y se puede demostrar por **reducción al absurdo:**

Supongamos que Greedy no es el óptimo.
Algoritmo óptimo elige conjunto O de contenedores, mientras que Greedy un conjunto G.

suma(O) > suma(G), ya que Greedy elige los más ligeros.

Pero entonces card(O) <= card(G), lo cual es una contradicción ya que Greedy no era el óptimo.

ALGORITMO QUE MAXIMIZA EL NÚMERO DE TONELADAS

Ordenamos el vector con los contenedores de mayor a menor peso y nos quedamos con los más pesados en el vector de salida hasta que se sobrepase la capacidad del buque.

Eficiencia teórica:
El algoritmo de Greedy es O(n) al recorrer el contenedor, pero al ordenar los pesos de mayor a menor cambia la eficiencia debido a que la función sort es O(n.logn).

```
void maxNumTons(int k, vector<double> & containers, vector<double> & containersResult)
{
  sort(containers.begin(), containers.end(), greater<double>() );

  // Capacidad de carga usada
  double capacityUsed = 0;

  for(int i=0; i!=containers.size(); ++i){

    if(capacityUsed+containers[i]<=k){
    // Añadimos el siguiente contenedor con menor peso
    containersResult.push_back(containers[i]);

    // Incrementamos la carga usada
    capacityUsed += containers[i];
    }
  }
```

ENFOQUE GREEDY

01	Conjunto de candidatos	• Contenedores disponibles
02	Conjunto de candidatos ya usados	• Contenedores que ya hemos metido en el barco
03	Función solución	• Nos indica cuando un subconjunto de contenedores es solución
04	Criterio de factibilidad	• Un conjunto de contenedores es válido mientras su peso total no exceda la carga máxima
05	Función de selección	• El siguiente candidato es el siguiente contenedor más pesado
06	Función objetivo	• Maximizar el peso total del barco

EJEMPLO DE EJECUCIÓN

```
Introduzca la capacidad de carga: 34
Introduzca los pesos de los contenedores: 1 2 3 4 5 6 7 8 9 -1
El máximo número de toneladas cargadas en el barco es: 34
Contenedores resultantes: 9 8 7 6 4
```

ESTUDIO DE LA OPTIMALIDAD

Como podemos observar en este ejemplo, el algoritmo de Greedy para este problema no es óptimo ya que no ha logrado maximizar el número de toneladas cargadas.

CONCLUSIONES DEL EJERCICIO 1

- El algoritmo Greedy no siempre es óptimo.
- La versión Greedy del algoritmo que maximiza el número de contenedores es óptima.
- La versión Greedy del algoritmo que maximiza el número de toneladas cargadas no resulta óptimo en todos los casos.

EJERCICIO 2

Problema del Viajante de comercio

HEURÍSTICA DEL VECINO MÁS CERCANO

Partiendo de una ciudad dada, v_0, añadimos como siguiente ciudad aquella con menor distancia de v_0 no incluida en el circuito. Seguimos con el mismo procedimiento hasta visitar todas las ciudades.

```
Node nearestNode(const Node & node,const vector<Node> & candidates,const vector<
vector<int>> & distances) {

  // Cogemos al primer candidato para comparar con él a los siguientes
  int minDist = distances[candidates[0].number-1][node.number-1];
  Node nearest = candidates[0];

  // Recorremos a todos los candidatos y nos quedamos con el más cercano
  for (auto elem : candidates) {
    int currDist = distances[elem.number-1][node.number-1];
    if (currDist < minDist) {
      minDist = currDist;
      nearest = elem;
    }
  }

  // Retornamos el más cercano
  return nearest;
}
```

Función auxiliar que selecciona, de entre los nodos posibles, el más cercano a uno dado

```
vector<Node> tspNearest(vector<Node> nodes)
{
  // Calculamos la matriz de distancias
  auto distances = distanceMatrix(nodes);
  vector<Node> result;

  // Metemos el primer nodo en la solución
  result.push_back(nodes[0]);
  removeNode(nodes, nodes[0]);

  // Bucle hasta que no queden nodos por visitar
  while (!nodes.empty()) {
    Node nearest = nearestNode(result.back(), nodes, distances);
    result.push_back(nearest);
    removeNode(nodes, nearest);
  }

  // Cerramos ciclo
  result.push_back(result[0]);

  return result;
}
```

INSERCIÓN MÁS ECONÓMICA

Pseudocódigo:

Función de selección: de entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito.

Complejidad: **$O(n^3)$**

```
Calculamos matriz de distancias;
Calculamos recorrido parcial;
mientras(quedan candidatos) {
        para (cada candidato de los candidatos) {
                para (cada nodo de la solución) {
                        Calculamos incremento al insertar el candidato;
                        Actualizamos la mejor posición en la que insertarlo;
                }
                Si el candidato es el mejor globalmente hasta ahora,
                pasa a ser el mejor;
        }
        Insertamos el mejor candidato en su posición en la solución;
        Lo eliminamos de los candidatos que quedan;
        Cerramos circuito;
}
```

INSERCIÓN MÁS LEJANA

Pseudocódigo:

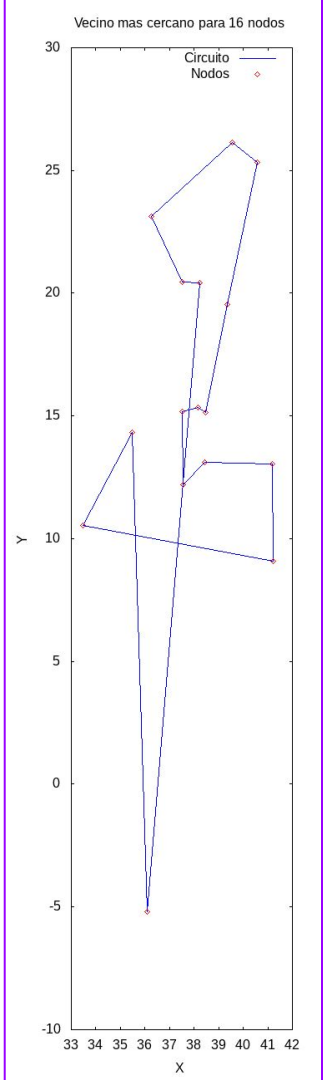
Función de selección: Se elige la ciudad no visitada más alejada del resto.
Se añade en entre las dos ciudades en las que provoque menor incremento en la longitud total del circuito.

Complejidad: **$O(n^3)$**

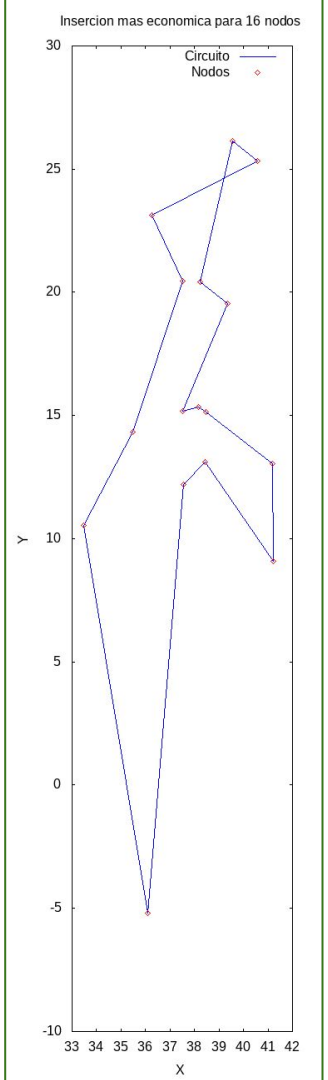
```
Calcula matriz de distancias;                                   // O(n^2)
Calcula recorrido parcial;        // Formado por los 2 nodos más lejanos, O(n^2)
mientras(quedan candidatos) {                                   // O(n)
        Calcula nodo más alejado como suma de distancias;       // O(n^2)

        para (cada nodo de la solución) {                       // O(n)
                Calculamos incremento al insertar el candidato;
                Actualizamos la mejor posición en la que insertarlo;
        }

        Insertamos el mejor candidato en su posición en la solución;
        Lo eliminamos de los candidatos que quedan;
        Cerramos circuito;
}
```

COMPARACIÓN DE LOS CIRCUITOS

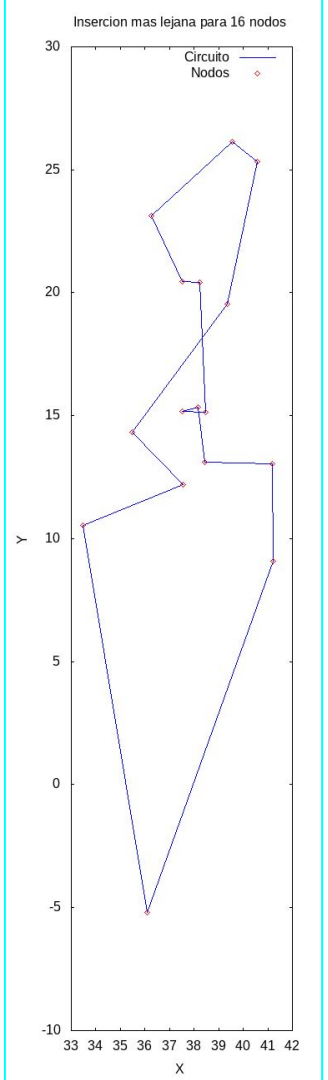
Vecino más cercano



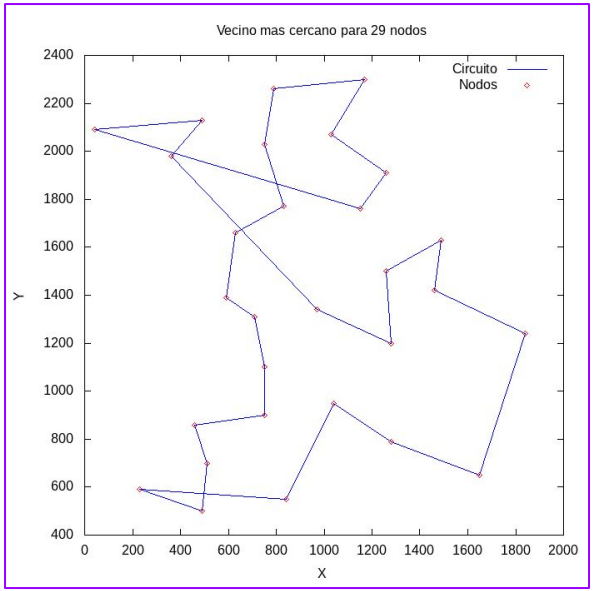
Inserción más económica



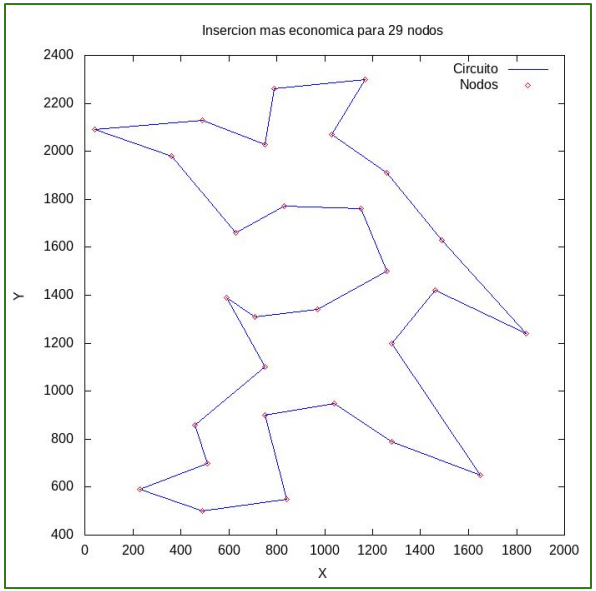
Inserción más lejana



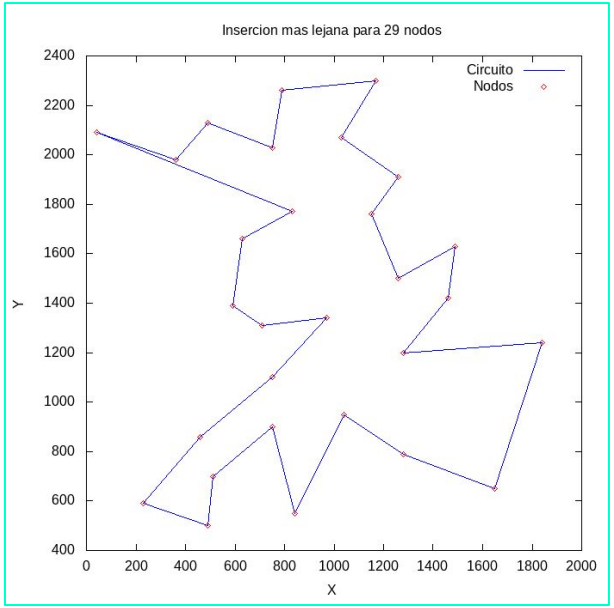
Vecino más cercano



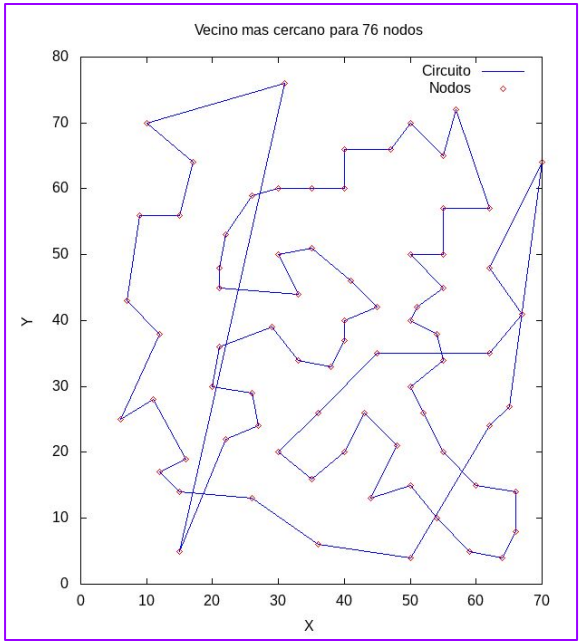
Inserción más económica



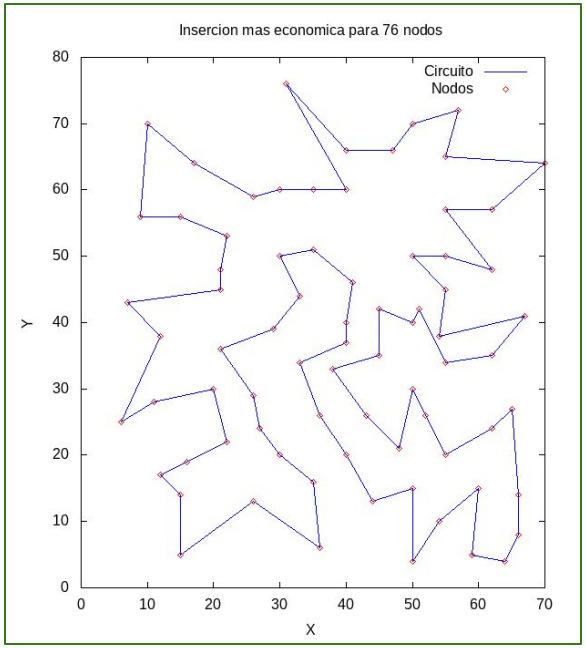
Inserción más lejana



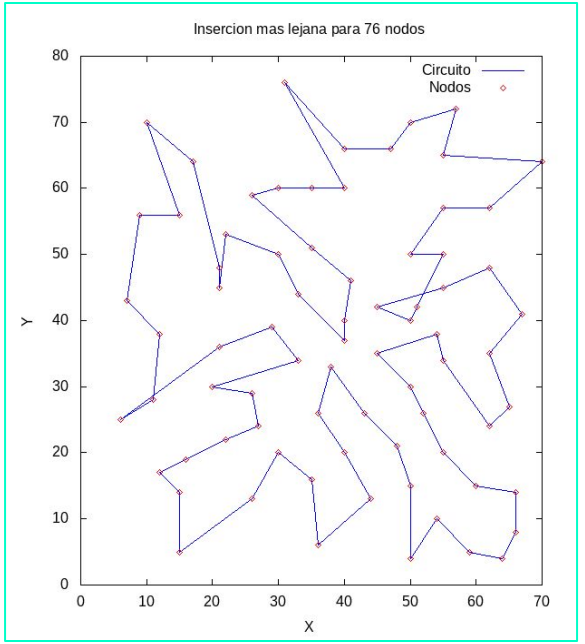
Vecino más cercano



Inserción más económica



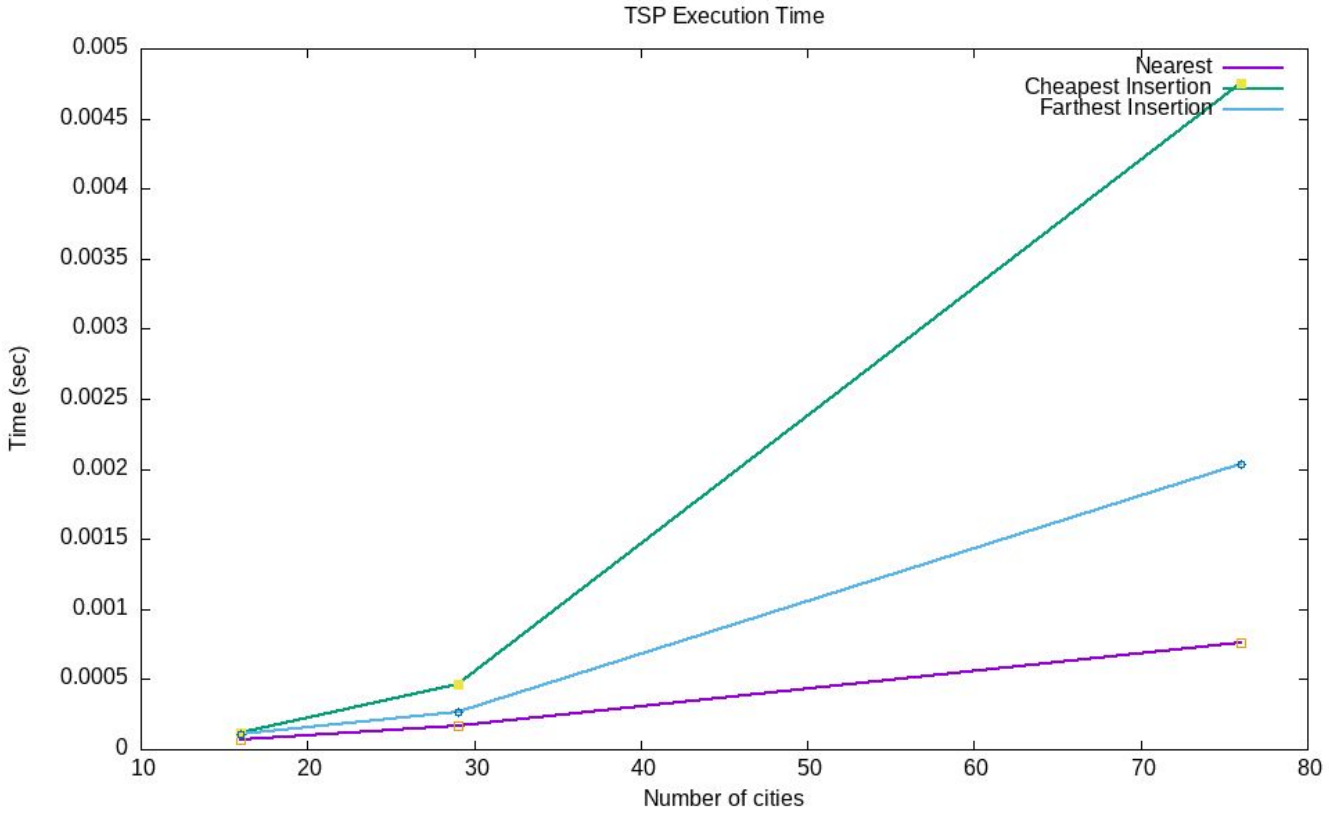
Inserción más lejana



COMPARACIÓN DE LA LONGITUD DEL CIRCUITO

	1. Más cercano	2. Inserción económica	3. Inserción lejana
ulysses16	79	70	66
bayg29	10200	9607	9476
eil76	662	575	586

COMPARACIÓN DE LOS TIEMPOS DE EJECUCIÓN



Conclusiones del Ejercicio 2

- En cuanto al algoritmo del vecino más cercano, se ha comprobado, en primer lugar, la trivialidad de la resolución, pues nace de la idea natural e intuitiva de ir a la posición que más cerca esté. Además, aunque es el algoritmo más eficiente (se ha observado que presenta una mayor velocidad de ejecución frente a los otros dos), se ha comprobado que no aporta la solución más óptima al problema; no reduce la distancia del recorrido tanto como los algoritmos de inserción.

- Por su parte, inserción lejana se ha comprobado que, preservando la eficiencia, es el algoritmo más óptimo de los tres; es el que consigue reducir más la distancia del recorrido, además de que presenta una mayor velocidad de ejecución frente al de inserción económica, que es el que mayores tiempos ha obtenido, pues, aunque consigue una reducción de la distancia mayor que el de cercanía, tiene una peor eficiencia.

- Orden de optimalidad:

  Inserción lejana > Inserción Económica > Cercanía

- Orden de eficiencia:

  Cercanía > Inserción Lejana > Inserción Económica