Universidad De Granada

E.T.S. de Ingenierías Informática y de Telecomunicación

Práctica 3: Algoritmos Voraces (Greedy)

Algorítmica

Autores:

Noura Lachhab Bouhmadi
Eduardo Rodríguez Cao
Ramón Liria Sánchez
Quintín Mesa Romero

Mayo 2022

ÍNDICE

0. INTRODUCCIÓN

En múltiples ocasiones cuando nos enfrentamos a un problema lo que nos interesa es, claramente, encontrar la mejor solución al mismo; somos algo codiciosos en cuanto a la búsqueda de la solución. Si lo único que perseguimos es hallar ansiadamente la solución del problema, en cada paso que damos en el proceso de resolución del mismo, elegimos la opción más beneficiosa en ese momento, sin mirar hacia el futuro; la elección depende solo del beneficio actual.

Paralelamente, en ciencias de la computación existen estrategias de búsqueda por las cuales se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Esto es lo que se conoce como Algoritmos Greedy (algoritmos golosos, ávidos, voraces…), que es de lo que se trata la práctica que estamos introduciendo.

El objetivo de la misma es que aprendamos a analizar un problema y resolverlo mediante la técnica Greedy, siendo capaces de justificar su eficiencia en términos de optimalidad.
Se han propuesto dos ejercicios para aplicar dicha técnica, los cuales se expondrán en puntos posteriores detalladamente y son:

- Contenedores
  - Maximizar el número de contenedores cargados y demostrar optimalidad.
  - Maximizar el número de toneladas cargadas.

- El problema del viajante de comercio
  - Heurística del vecino más cercano.
  - Inserción.
  - Heurística propia del grupo.

En el primer ejercicio, en primer lugar se mostrará el código de cada uno de los algoritmos, a partir del cual se obtendrá la eficiencia teórica de cada uno. Además, se mostrará un ejemplo de ejecución de cada uno así como también se llevará a cabo un estudio de la optimalidad de los mismos, para finalmente sacar una serie de conclusiones.

Por su parte, en el segundo, en primer lugar se mostrarán las funciones auxiliares comunes a cada una de las heurísticas abordadas, para a continuación proceder a la exposición detallada de las mismas. Para cada algoritmo se ha estudiado la eficiencia teórica. Además, se llevará a cabo una comparativa entre las longitudes del recorrido usando los tres algoritmos, así como también se hará una comparativa entre los distintos tiempos de ejecución para finalmente sacar conclusiones al respecto.

1. LISTADO DEL HARDWARE UTILIZADO EN LA PRÁCTICA

A continuación se presenta un listado de los dispositivos, junto con sus características, con los que se ha llevado a cabo esta práctica.

- ❖ Ordenador de Noura Lachhab Bouhmadi:
  - ☐ **Nombre del dispositivo**: GF63-Thin-10SCXR
  - ☐ **CPU:** Intel® Core™ i7-10750H CPU @ 2.60GHz × 12
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** Intel® UHD Graphics (CML GT2)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Eduardo Rodríguez Cao:
  - ☐ **Nombre del dispositivo**: Acer Nitro 5 AN515
  - ☐ **CPU:** Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4
  - ☐ **Memoria RAM:** 12 GB
  - ☐ **Tarjeta gráfica:** NVIDIA GeForce GTX 1050 Ti
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Ramón Liria Sánchez:
  - ☐ **Nombre del dispositivo**: Asus Rog Strix G513IH-HN008
  - ☐ **CPU:** AMD Ryzen 7 4800H (8MB Cache, 2.9GHz)
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** NVIDIA GeForce GTX 1650 (4GB GDDR6)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS
- ❖ Ordenador de Quintín Mesa Romero:
  - ☐ **Nombre del dispositivo**: Lenovo-Yoga-S740-14IIL
  - ☐ **CPU:** Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
  - ☐ **Memoria RAM:** 16 GB
  - ☐ **Tarjeta gráfica:** Intel® Iris(R) Plus Graphics (ICL GT2)
  - ☐ **Sistema operativo:** Ubuntu 20.04.3 LTS

2. SOFTWARE UTILIZADO

A continuación se especifica todo lo relativo al software utilizado en la práctica.

- Para la **edición del código de los algoritmos** (programas escritos en el lenguaje de programación C++), se ha empleado el editor de textos que por defecto viene instalado en Linux:

- Para la **compilación de los programas** se ha utilizado el compilador de *línea de órdenes* que compila y enlaza programas en C++, **g++**.
- La **generación de gráficas** se ha llevado a cabo mediante **gnuplot**, usando las correspondientes órdenes especificadas en el guión de la práctica.
- Para la elaboración de la memoria de la práctica se ha utilizado la herramienta **google docs**.

3. EJERCICIO 1: Contenedores

Se tiene un buque mercante cuya capacidad de carga es de K toneladas y un conjunto de contenedores $c_1, \ldots, c_n$ cuyos pesos respectivos son $p_1, \ldots, p_n$ (expresados también en toneladas). Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores:

1.Diseñe un algoritmo que maximice el número de contenedores cargados, y demuestre su optimalidad.

2.Diseñe un algoritmo que intente maximizar el número de toneladas cargadas.

3.1 Apartado 1

3.1.1 Algoritmo que maximiza el número de contenedores cargados

El algoritmo Greedy para este problema tiene los siguientes elementos:

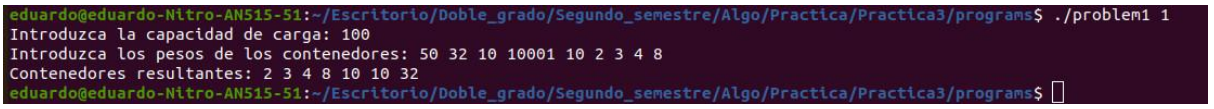
1. Conjunto de candidatos: los contenedores disponibles.
2. Conjunto de candidatos ya usados: los contenedores que ya hemos metido en el barco.
3. Función solución: cuando se sobrepasa la capacidad de carga del buque.
4. Criterio de factibilidad: mientras no se sobrepase la capacidad de carga.
5. Función de selección: el contenedor más ligero.
6. Función objetivo: maximizar el número de contenedores cargados.

El código en c++ es el siguiente:

```
/**
 * @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];
  }
}
```

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.

3.1.2 Ejemplo de ejecución



3.1.3 Estudio de la optimalidad

El algoritmo Greedy en este problema es óptimo, para demostrarlo usaremos reducción al absurdo:

Supongamos que el algoritmo Greedy no es el óptimo. Entonces el algoritmo óptimo elegirá un conjunto O de contenedores, mientras que el Greedy un conjunto G. suma(O) > suma(G), ya que el óptimo al no ser el Greedy no elegirá a los contenedores más ligeros, luego la suma de los pesos de los contenedores de O será estrictamente mayor que la de G.

Pero entonces card(O) <= card(G), lo cual no es posible ya que hemos dicho que el algoritmo Greedy no es el óptimo. Al llegar a una contradicción tenemos que el algoritmo Greedy es el óptimo.

3.1.4 Análisis de la eficiencia teórica

El algoritmo Greedy es claramente O(n) al recorrer el contenedor. Si tuviéramos el contenedor ya ordenado sería efectivamente O(n), pero al tener que ordenarlo primero es O(nlogn).

3.2 Apartado 2

3.2.1 Algoritmo que maximiza el número de toneladas cargadas

Para crear este algoritmo, vamos a enfocarnos en el algoritmo Greedy: Trataremos de ir incluyendo los contenedores más pesados hasta que no quepan más. Los elementos que nos definen el enfoque Greedy son:

1. Conjunto de candidatos: Son los contenedores disponibles.
2. Conjunto de candidatos ya usados: Son los contenedores que ya hemos metido en el barco.
3. Función solución: que nos indica cuando un subconjunto de candidatos forma una solución cuando no quepa ninguno de los contenedores restantes debido a la capacidad máxima del barco.
4. Criterio de factibilidad: Un conjunto de contenedores es válido mientras que su peso total no exceda la carga máxima.
5. Función de selección:El siguiente candidato es el siguiente contenedor más pesado.
6. Función objetivo: El objetivo de este algoritmo es maximizar el peso total del barco.

Aplicando todos estos criterios a nuestro algoritmo, obtenemos lo siguiente:

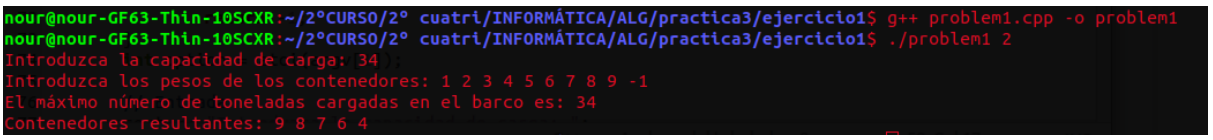
```cpp
void maxNumTons(int k, vector<double> & containers, vector<double> & containersResult)
{
  sort(containers.begin(), containers.end(), greater<double>() );

  // 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];
    }
  }


}
```

Este método lo que hace es ordenar los contenedores de los más pesados a los menos pesados e ir insertándolos en el barco. Y mediante la variable capacityUsed vamos controlando el peso de los contenedores que vamos introduciendo para que no excedamos la carga máxima que puede soportar el barco.

3.2.2 Ejemplo de ejecución

Como podemos observar en la captura de pantalla, le pasamos a nuestro programa los pesos de los contenedores de los que disponemos hasta que le insertamos -1. A continuación ordena esos pesos de mayor a menor y los va insertando en el barco hasta que ya no quepa más toneladas.

En este caso, vemos que los contenedores insertados en el barco son 5 y con un peso de 9, 8, 7, 6 y 4 toneladas, y de esta manera hemos logrado aprovechar toda la capacidad del barco ya que la suma de los contenedores introducidos es de 34 toneladas que es exactamente la capacidad del barco.



3.2.3 Estudio de la optimalidad

El algoritmo Greedy para este problema no es óptimo. Para probarlo, pongamos un contraejemplo:
Supongamos que tenemos un buque con capacidad máxima de 11 toneladas y una serie de contenedores de pesos: 8, 5 y 6 toneladas.
Los contenedores resultantes al aplicar el algoritmo Greedy son: 8.

Claramente, no se ha logrado maximizar el número de toneladas cargadas, pues, habiendo cargado los dos contenedores de 5 y 6 toneladas respectivamente, se lograría cargar un mayor tonelaje que cargando solo el contenedor de 8 toneladas, como se obtiene al aplicar Greedy.

3.2.4 Análisis de la eficiencia teórica

Si observamos el código del algoritmo vemos que consta en primer lugar de una llamada a la función sort de la stl cuya eficiencia es O(nlogn) y, a continuación un bucle for que se ejecuta n veces en el peor de los casos (n≡tamaño del vector con los pesos de los contenedores), cuyo cuerpo consta de sentencias simples que suponen tiempos constantes (ya que la función de push_back() de la stl tiene una eficiencia de O(1)). Con lo cual, podemos afirmar por la regla de la suma que la eficiencia del algoritmo, es O(nlogn).

3.3 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.

4. EJERCICIO 2: El problema del viajante de comercio

El problema del viajante de comercio (TSP, por Traveling Salesman Problem) consiste en lo siguiente: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Explicado de una forma más formal, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

Para resolver el problema se han abordado tres heurísticas distintas:

- *Vecino más cercano*
- *Inserción más económica*
- *Inserción más lejana*

Todas ellas se exponen detalladamente a continuación:

4.0 Funciones auxiliares comunes a todas las heurísticas

Para resolver este problema hemos usados funciones auxiliares que son comunes para todas las heurísticas:

- struct Node: estructura que representa un nodo del grafo, con los siguientes datos miembro: número del nodo, coordenada x y coordenada y

```
/**
 * @brief Estructura que representa un nodo del grafo. Los miembros son:
 * el número del nodo, la coordenada x y la coordenada y
 */
struct Node {
  int number;
  double x;
  double y;
};
```

- distance: distancia euclídea truncada entre dos nodos dados, es un cálculo **O(1)**.

```
/**
 * @brief Función que devuelve un entero con la distancia entre dos nodos dados
 * @param n1 Primer nodo
 * @param n2 Segundo nodo
 * @return Entero con la distancia euclídea entre los nodos dados
 */
int distance(const Node & n1, const Node & n2) {
  // Distancia euclídea truncada
  return trunc(sqrt(pow(n2.x-n1.x,2) + pow(n2.y-n1.y,2)));
}
```

- distanceMatrix: calcula la matriz de distancias para los nodos dados, es un cálculo **O(n^2)**, debido a que tiene dos bucles **O(n)** anidados que usan la función **O(1)** distance.

```
/**
 * @brief Función que calcula la matriz con las distancias entre los nodos
 * @param nodes Vector con los nodos
 * @return Matriz con las distancias entre nodos
 */
vector<vector<int>> distanceMatrix(const vector<Node> & nodes) {
  // Vector salida de dimensiones size x size
  int size = nodes.size();
  vector<vector<int>> result(size, vector<int>(size));

  // Lo rellenamos con las distancias
  for (int i = 0; i < size; ++i) {
    for (int j = i; j < size; ++j) {
      int dist = distance(nodes[i], nodes[j]);
      result[i][j] = dist;
      result[j][i] = dist;
    }
  }
  return result;
}
```

- removeNode: elimina el nodo dado del vector de nodos dado, es **O(n)** por recorrer el vector de nodos dado.

```
/**
 * @brief Función que elimina el nodo dado del vector de nodos dado
 * @param nodes Vector con los nodos, será modificado
 * @param node Nodo a eliminar
 */
void removeNode(vector<Node> & nodes, const Node & node) {
  bool deleted = false;
  auto it = nodes.begin();
  // Recorremos el vector hasta el final o hasta encontrar
  // el nodo a eliminar.
  while (!deleted && it != nodes.end()) {
    if (it->number == node.number) {
      nodes.erase(it);
      deleted = true;
    }
    it++;
  }
}
```

- routeLength: calcula la longitud del circuito, es **O(n)** por recorrer el vector de nodos.

```
/**
 * @brief Función que devuelve la longitud del camino entre los nodos
 * @param nodes Vector con los nodos del camino
 * @return Entero con la longitud del camino
 */
int routeLength(const vector<Node> & nodes) {
  int size = nodes.size();
  int length = 0;
  for (int i = 1; i < size; ++i)
    length += distance(nodes[i-1], nodes[i]);
  return length;
}
```

4.1 Vecino más cercano

Este es el primer algoritmo que se propone para la resolución del problema. La heurística del vecino más cercano funciona de la siguiente manera: partiendo de una ciudad dada, v_0, añadimos como siguiente ciudad aquella que se encuentre a menor distancia de v_0. Dicha distancia no ha de estar incluida en el circuito. Seguimos con el mismo procedimiento hasta que todas las ciudades se hayan visitado.

El código del algoritmo que se propone es el siguiente:

```
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;
}
```

Algoritmo del vecino más cercano

Analicemos **teóricamente** la **eficiencia** de este algoritmo:
Como bien podemos observar, si nos situamos al principio del código, lo que nos encontramos es una sentencia en la que se hace una llamada a la función *distanceMatrix*, la cual supone un tiempo cuadrático, porque consta de dos bucles anidados que se ejecutan n veces cada uno, luego tiene una eficiencia $O(n^2)$. A continuación, se hace una llamada a la función *push_back* de la clase vector de la stl, la cual supone un tiempo constante (O(1)), y seguidamente a la función *removeNode* (función auxiliar común a todas las heurísticas, mostrada anteriormente), que supone una tiempo lineal (O(n)). Avanzando en el código nos topamos con un bucle while que se ejecuta n veces y, dentro del mismo, tres sentencias en las que se llaman a las dos funciones anteriormente mencionadas, y a una función auxiliar, *nearestNode*, que tiene una eficiencia O(n) luego, el cuerpo del bucle, por la regla de la suma supone un tiempo lineal (O(n)) y como se ejecuta n veces, $O(n^2)$. Finalmente, fuera del bucle, se llama a push_back; O(1).
Por lo tanto, aplicando la regla de la suma, podemos afirmar que la eficiencia del algoritmo de cercanía es **$O(n^2)$**.

4.2 Inserción más económica

En las estrategias de inserción, la idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes mediante algún criterio de tipo greedy. Para poder implementar este tipo de estrategia, deben definirse tres elementos:

1. Cómo se construye el recorrido parcial inicial.
2. Cuál es el nodo siguiente a insertar en el recorrido parcial.
3. Dónde se inserta el nodo seleccionado.

El recorrido inicial se puede construir a partir de las tres ciudades que formen un triángulo lo más grande posible: por ejemplo, eligiendo la ciudad que está más al Este, la que está más al Oeste, y la que está más al norte.
Cuando se haya seleccionado una ciudad, esta se ubicará en el punto del circuito que provoque el menor incremento de su longitud total. Es decir, hemos de comprobar, para cada posible posición, la longitud del circuito resultante y quedarnos con la mejor alternativa.
Por último, para decidir cuál es la ciudad que añadiremos a nuestro circuito, podemos aplicar el siguiente criterio, denominado inserción más económica: de entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito.
En otras palabras, cada ciudad debemos insertarla en cada una de las soluciones posibles y quedarnos con la ciudad (y posición) que nos permita obtener un circuito de menor longitud.
Seleccionaremos aquella ciudad que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades.

El código de la heurística se apoya en las funciones auxiliares comunes y otras nuevas como:

- initialRoute: calcula los 3 nodos del recorrido parcial y es **O(n)** al recorrer todos los nodos.

```
/**
 * @brief Función que calcula el recorrido parcial y elimina esos nodos del vect
 * or inicial
 * @param nodes Vector con los nodos
 * @return Vector con los nodos del recorrido parcial, se modifica el vector ini
 * cial
 */
vector<Node> initialRoute(vector<Node> & nodes) {
  vector<Node> result;
  Node north = nodes[0], east = nodes[0], west = nodes[0];

  // Calculamos los nodos más al norte, este y oeste
  for (auto node : nodes) {
    if (node.y > north.y)
      north = node;
    if (node.x > east.x)
      east = node;
    if (node.x < west.x)
      west = node;
  }

  // Rellenamos el recorrido parcial
  result.push_back(north);
  result.push_back(east);
  result.push_back(west);

  // Los eliminamos del vector original
  removeNode(nodes, north);
  removeNode(nodes, east);
  removeNode(nodes, west);

  return result;
}
```

- lengthIncrease: calcula el coste de añadir un nodo en la solución entre dos nodos de la solución. Es decir, calcula a+b-c, donde a y b son las distancias entre el nodo a insertar y los nodos entre los que se inserta, mientras que c es la distancia entre los nodos de la solución entre los que se inserta. Es un cálculo **O(1)** que usa la matriz de distancias.

```
/**
 * @brief Función que calcula el incremento en la longitud al insertar el nodo
 * con índice k entre los nodos con índices i y j
 * @param k nodo a insertar
 * @param i uno de los nodos entre los que se inserta
 * @param j uno de los nodos entre los que se inserta
 * @param distances Matriz con las distancias entre nodos
 * @return Entero con el incremento en longitud
 */
int lengthIncrease(int k, int i, int j, const vector<vector<int>> & distances) {
  return distances[k][i] + distances[k][j] - distances[i][j];
}
```

El código de la heurística es el siguiente:

```
/**
 * @brief Función que dado un vector con los nodos de entrada devuelve
 * un vector de salida con los nodos reordenados por la heurística de inserción
 * @param input Vector con los nodos de entrada
 * @return Vector con los nodos reordenados
 */
vector<Node> tspCheapestInsertion(vector<Node> nodes) {

  // Calculamos la matriz de distancias y el recorrido parcial
  auto distances = distanceMatrix(nodes);
  auto result = initialRoute(nodes);

  // Bucle hasta que no queden nodos por visitar
  while (!nodes.empty()) {

    // Empezamos con un primer candidato para comparar con él
    Node bestCandidate = nodes[0];
    int bestIncrease = lengthIncrease(bestCandidate.number-1, result[0].number-
1, result[1].number-1, distances);
    auto bestInsertionIter = result.begin() + 1;
    int resultSize = result.size();

    // Recorremos todos los nodos por visitar
    for (auto candidate : nodes) {

      // Incremento en distancia al insertar el candidato entre
      // el primer y el último nodo del recorrido actual
      // Si es el mínimo para el candidato se insertará al principio del result
      int minIncrease = lengthIncrease(candidate.number-1, result[0].number-1, r
esult[resultSize-1].number-1, distances);
      auto insertionIter = result.begin();

      // Recorremos los nodos del recorrido actual
      for (int i = 1; i < resultSize; ++i) {

        // Incremento en distancia al insertar el candidato entre el nodo
        // i-1 y i del recorrido actual
        int increase = lengthIncrease(candidate.number-1, result[i-1].number-1,
result[i].number-1, distances);
```

```
        // Si el incremento es menor que el mínimo, lo actualizamos
        if (increase < minIncrease) {
          minIncrease = increase;
          insertionIter = result.begin() + i;
        }
      }

      // Si el candidato es el mejor hasta ahora, actualizamos el mejor
      if (minIncrease < bestIncrease) {
        bestCandidate = candidate;
        bestIncrease = minIncrease;
        bestInsertionIter = insertionIter;
      }
    }

    // Insertamos el mejor candidato en la solución y lo eliminamos de los nodos
candidatos
    result.insert(bestInsertionIter, bestCandidate);
    removeNode(nodes, bestCandidate);
  }

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

  return result;
}
```

Calculamos la matriz de distancias (**O(n^2)**) y el recorrido parcial (**O(n)**). Después empezamos el algoritmo Greedy. Mientras nos quedan candidatos, en un bucle recorremos todos los candidatos que todavía no están en la solución. Para cada uno de ellos en otro bucle recorremos los nodos que ya están en la solución y calculamos la posición en la que insertaríamos ese nodo candidato si fuera el óptimo, minimizando el coste de insertarlo en la ruta actual. Fuera de ese bucle comprobamos si ese candidato es mejor que el candidato más óptimo hasta ese momento, si lo es pasa a ser el candidato más óptimo. Una vez que obtenemos el candidato óptimo definitivo, lo eliminamos de los candidatos y lo añadimos a la solución en la posición precalculada. El proceso se repite hasta que no queden candidatos.
Por lo tanto, al tener tres bucles **O(n)** anidados que hacen llamadas a funciones auxiliares **O(1)**, la heurística es **O(n^3)**, ya que por la regla de la suma n^3 domina a n^2 y n.

4.3 Inserción más lejana

Como última estrategia se ha escogido inserción más lejana, la cual comienza con un recorrido inicial y a continuación inserta el resto de ciudades según cierto criterio. Al igual que inserción más económica, consta de tres elementos a definir:

1. Cómo se construye el recorrido parcial inicial.
2. Cuál es el nodo siguiente a insertar en el recorrido parcial.

3. Dónde se inserta el nodo seleccionado.

El recorrido inicial está formado por las dos ciudades con mayor distancia entre sí. A continuación, se selecciona la ciudad cuya suma de distancias a cada una de las ciudades del recorrido formado es mayor. Esta se insertará en el punto del circuito que menos incremente su longitud, por lo que se comprobará que aumento causará al añadirla entre cada dos ciudades del circuito ya formado.
Puede parecer contraintuitivo insertar las ciudades cuya distancia al resto es mayor, pero este algoritmo puede llegar a ser mejor que el de cercanía e inserción económica en bastantes ocasiones.

El código que ejecuta el algoritmo se apoya en estas funciones auxiliares:

- initialRouteFarthest: calcula los 2 nodos del recorrido parcial, siendo estos los dos cuya distancia entre sí sea mayor. Es **O(n^2)** al recorrer la matriz de distancias entre los nodos, donde buscará la mayor distancia entre dos de ellos.

```
/**
 * @brief Función que calcula los 2 nodos del recorrido parcial, siend
 *  o estos los dos cuya distancia entre sí sea mayor
 * @param nodes Vector con los nodos
 * @param distances Vector con las distancias
 * @return Vector con los nodos del recorrido parcial más lejanos.
*/

vector<Node> initialRouteFarthest(vector<Node> & nodes, const vector<vector<int>
> & distances){
  // La ruta inicial está formada por los nodos más lejanos
  Node a, b;
  int farthestDistance = distances[0][0];

  // Busca los dos nodos más lejanos entre sí
  for (int i=0; i<distances.size(); i++)
    for (int j=0; j<distances[i].size(); j++)
      if (distances[i][j] > farthestDistance){
        farthestDistance = distances[i][j];
        a = nodes[i];
        b = nodes[j];
      }

  vector<Node> result;
  result.push_back(a);
  removeNode(nodes, a);
  result.push_back(b);
  removeNode(nodes,b);

  return result;
}
```

- farthestNode: Función que calcula el punto más alejado del circuito teniendo en cuenta la suma de las distancias del nodo a cada uno de los que forman el circuito. Es **O(n^2)** ya que recorre dos bucles anidados, los candidatos posibles y la distancia de estos a cada uno de los nodos del circuito.

```
/**
 * @brief Función que calcula el punto más alejado del circuito teniendo en cuen
ta
 * la suma de las distancias del nodo a cada uno de los que forman el circuito
 * @param circuit Circuito con al que se calcularán las distancias a cada nodo
 * @param candidates Nodos candidatos
 * @param distances Matriz con las distancias entre nodos
 * @return Nodo más lejano al circuito
 */
Node farthestNode(const vector<Node> & circuit, const vector<Node> & candidates,
const vector<vector<int>> & distances){
  // Empezamos con un primer candidato para comparar con él
  Node farthestCandidate = candidates[0];
  int farthestDistance = distances[farthestCandidate.number-1][circuit[0].number
-1];

  // Recorremos todos los nodos por visitar
  for (auto candidate : candidates) {

    int distance = 0;

    // Suma de distancias entre el candidato y cada nodo insertado
    for (int i = 0; i < circuit.size(); ++i) {
      distance += distances[candidate.number-1][circuit[i].number-1];
    }

    // Si el candidato es el mejor hasta ahora, actualizamos el mejor,
    // es decir, el que maximice distance
    if (distance > farthestDistance) {
      farthestCandidate = candidate;
      farthestDistance = distance;
    }
  }

  return farthestCandidate;
}
```

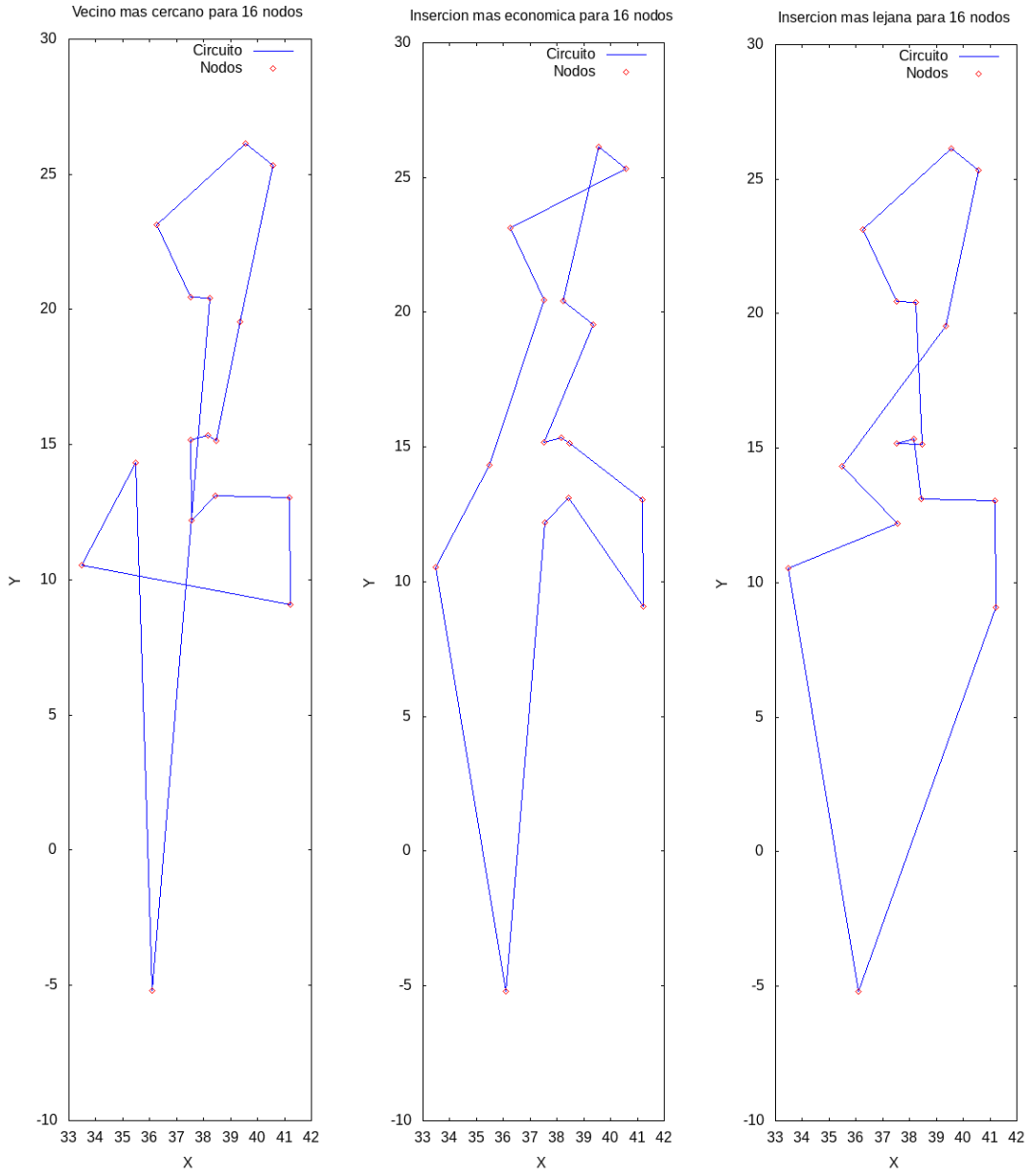
El código de la heurística es el siguiente:

```
/**
 * @brief Función que dado un vector con los nodos de entrada devuelve
 * un vector de salida con los nodos reordenados por la heurística de inserción
lejana.
 * Inserta los nodos que maximicen su mínimo coste al añadirse, al contrario de
 * inserción, que añader los que minimizan su mínimo coste al añadirse.
 * @param input Vector con los nodos de entrada
 * @return Vector con los nodos reordenados
 */
vector<Node> tspFarthestInsertion(vector<Node> nodes) {

  // Calculamos la matriz de distancias y el recorrido parcial
  auto distances = distanceMatrix(nodes);
  auto result = initialRouteFarthest(nodes, distances);


  // Bucle hasta que no queden nodos por visitar
  while (!nodes.empty()) {
    // Buscamos el candidato a insertar
    Node bestCandidate = farthestNode(result, nodes, distances);
```

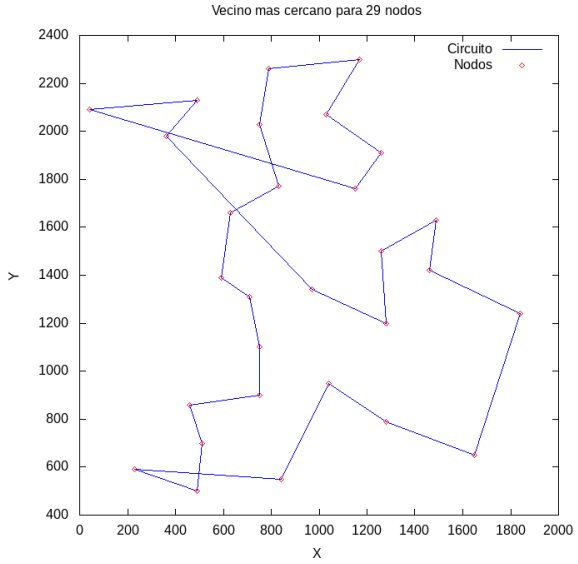
```
    // Buscamos la posición donde se insertará el candidato elegido

    // Empezamos con el incremento en distancia al insertar el candidato entre
    // el primer y el último nodo del recorrido actual
    int resultSize = result.size();
    int minIncrease = lengthIncrease(bestCandidate.number-1, result[0].number-1,
result[resultSize-1].number-1, distances);
    auto insertionIter = result.begin();

    // Recorremos los nodos del recorrido actual
    for (int i = 1; i < resultSize; ++i) {

      // Incremento en distancia al insertar el candidato entre el nodo
      // i-1 y i del recorrido actual
      int increase = lengthIncrease(bestCandidate.number-1, result[i-1].number-
1, result[i].number-1, distances);

      // Si el incremento es menor que el mínimo, lo actualizamos
      if (increase < minIncrease) {
        minIncrease = increase;
        insertionIter = result.begin() + i;
      }
    }

    // Insertamos el mejor candidato en la solución y lo eliminamos de los nodos
candidatos
    result.insert(insertionIter, bestCandidate);
    removeNode(nodes, bestCandidate);
  }

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

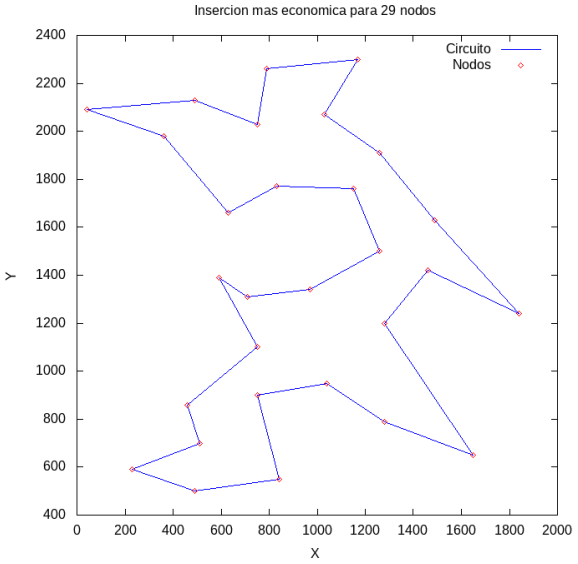
Calculamos la matriz de distancias (**O(n²)**) y el recorrido inicial con los dos nodos más lejanos (**O(n²)**).
Después entramos en un bucle por cada uno de los nodos a insertar. En este buscamos cuál será el próximo nodo que se añada gracias a la función farthestNode (**O(n²)**), que encontrará el nodo más lejano. A continuación, sin salir del bucle, se realiza otro bucle (**O(n)**) en el que se calcula la posición a insertar el nodo para que el aumento de distancia en el recorrido sea mínimo. Por último se añade el candidato al resultado y se elimina de los candidatos. En estos últimos pasos, por la regla de la suma **O(n²)** dominará a (**O(n)**), que al estar anidado en el bucle general, este será (**O(n³)**).
Para la eficiencia del método vemos como este último bucle con (**O(n³)**) domina a la creación del recorrido inicial con (**O(n²)**), por tanto la heurística cuenta con eficiencia (**O(n³)**).

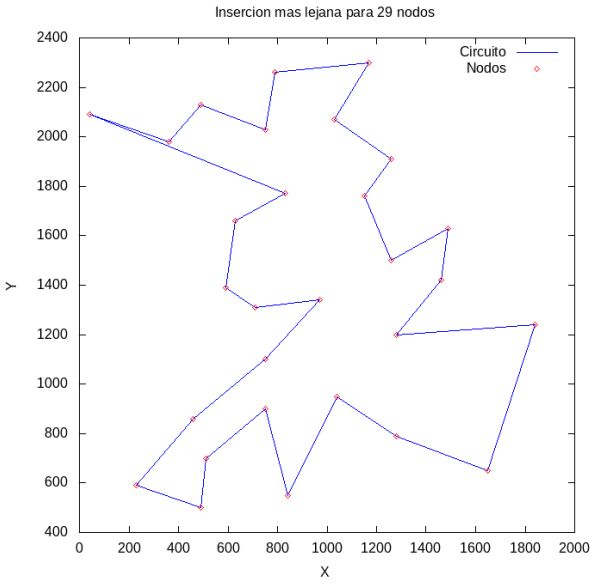
4.4 Comparación de los circuitos



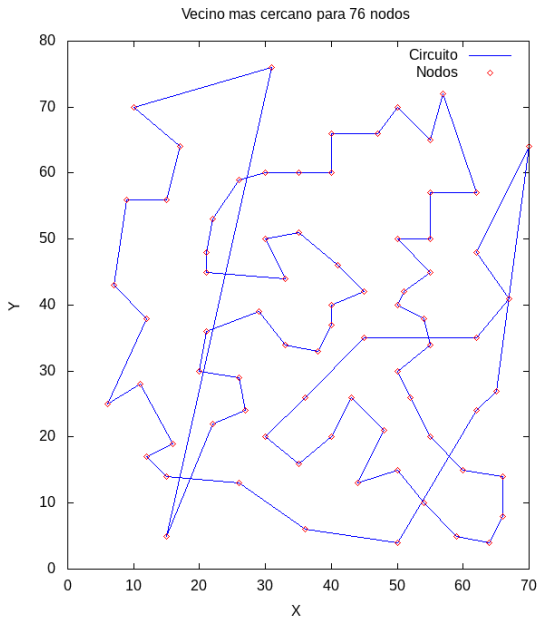
Como una observación podemos decir que las heurísticas de inserción encuentran solución más óptima que la heurística del vecino más cercano. A primera vista se ve como la de cercanía puede llegar a hacer un viaje muy largo porque el último nodo por visitar está muy lejos. Mientras que las de inserción se cruzan poco y parecen ser más razonables al "extenderse" entre los nodos.

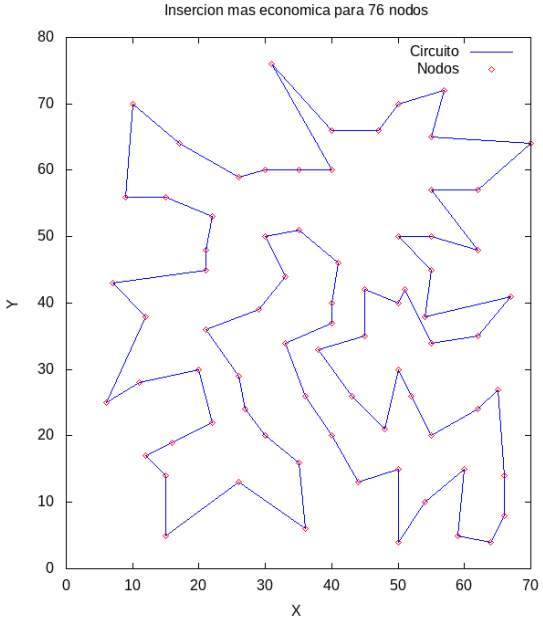


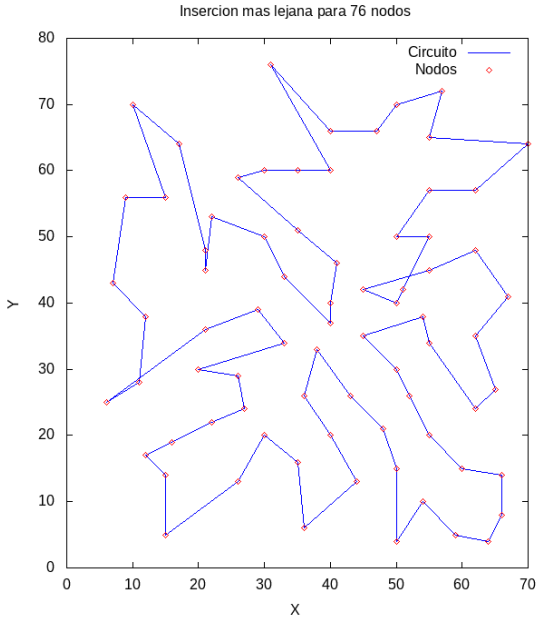




Esta observación se ve más clara para mayor número de nodos.







No obstante, no se ve gran diferencia entre la inserción más económica y la inserción lejana. Al tener solamente 3 datos de prueba no se puede ver empíricamente cuál es mejor, pero según estudios hechos la inserción lejana suele ser mucho mejor que la inserción económica.

4.5 Comparación de la longitud del circuito

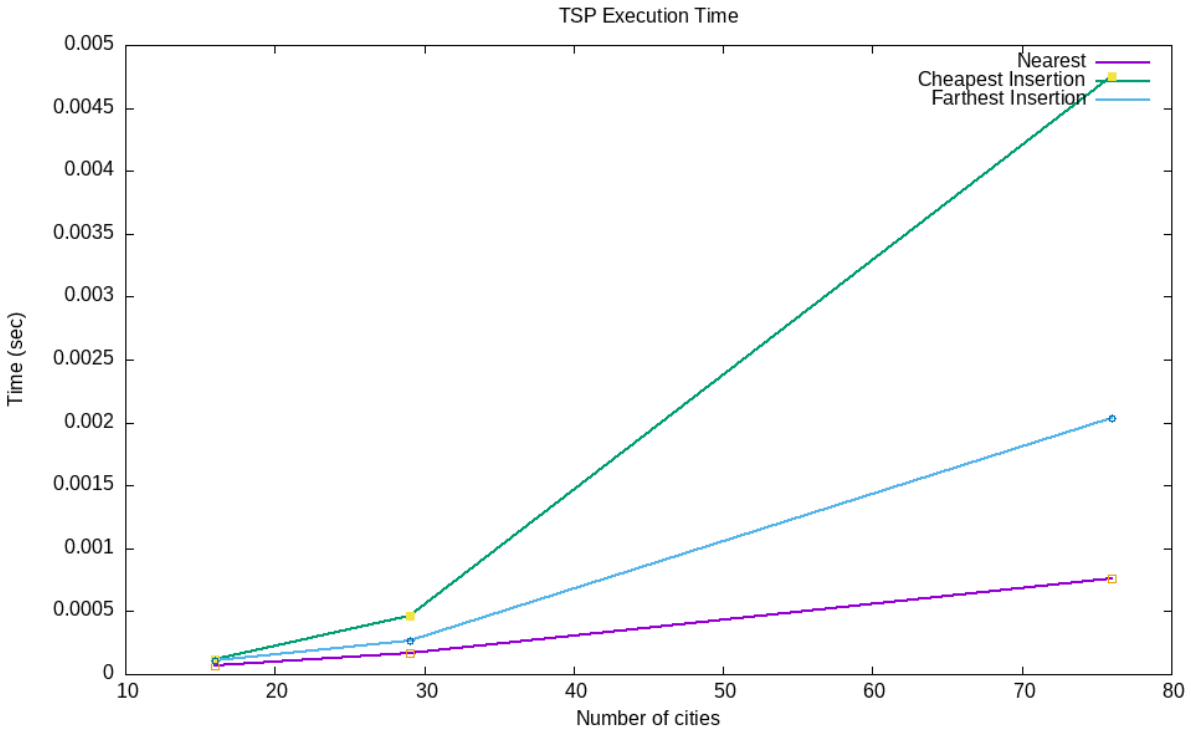
Vemos la distancia que tiene que recorrer el viajante de comercio para los recorridos generados por cada uno de los algoritmos en los tres casos distintos de grupos de ciudades.

	1. Más cercano	2. Inserción económica	3. Inserción lejana
ulysses16	79	70	66

bayg29	10200	9607	9476
eil76	662	575	586

Observamos como el algoritmo de cercanía, aunque es el más eficiente, no logra encontrar el mejor recorrido en ninguno de los casos. Inserción económica reduce la distancia del recorrido, a coste de empeorar la eficiencia. Inserción lejana mantiene su eficiencia pero logra mejorar la distancia recorrida en dos casos, por lo que da mejores resultados, a pesar de la idea contraintuitiva de introducir los nodos más lejanos primero.

4.6 Comparación de los tiempos de ejecución



Vemos en los tiempos de ejecución como el algoritmo por cercanía es el más rápido en los tres datos de prueba debido a su eficiencia $O(n^2)$. Tanto inserción económica como lejana son $O(n^3)$, lo que como observamos aumenta su tiempo. Sin embargo, con la misma eficiencia, inserción lejana es más rápido que inserción económica en todos los casos, obteniendo además mejores resultados en dos de ellos, como hemos visto antes. Por tanto, el tercer algoritmo o inserción lejana puede que sea la mejor elección a la hora de elegir uno de ellos.

4.7 Conclusiones del ejercicio

- 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