Práctica 3
Algoritmos Greedy
UGR - ETSIIT - ALGORÍTMICA - B1

Universidad de Granada

Componentes del Grupo "Oliva"

Jose Luis Pedraza Román

Pedro Checa Salmerón

Antonio Carlos Perea Parras

Raúl Del Pozo Moreno

Índice:

1. **Descripción de los problemas**
2. **Hardware y software utilizado**
3. **Travelling Salesman Problem (TSP)**
    - 3.1. **Cercanía**
    - 3.2. **Inserción**
    - 3.3. **Propio → 2-opt**
    - 3.4. **Ficheros de prueba**
    - 3.5. **Comparativa**
4. **Recubrimiento de un grafo no dirigido**
    - 4.1. **Grafo**
        - 4.1.1. **Demostración de no minimalidad en grafos**
        - 4.1.2. **Pruebas de ejecución**
    - 4.2. **Árbol**
        - 4.2.1. **Pruebas de ejecución**
        - 4.2.2. **Ejemplo de ejecución**
    - 4.3. **Ficheros de prueba**
    - 4.4. **Comparativa**
5. **Conclusiones**
6. **Bibliografía**

1. Descripción de los problemas

En esta práctica vamos a resolver dos problemas, el problema del viajante de comercio mediante tres heurísticas diferentes (vecino más cercano, inserción y una propia) mediante los cuales se obtendrán unas rutas por las que deberá pasar el viajante de forma que la ruta tenga la menor distancia posible, y el problema de recubrimiento de un grafo no dirigido, en el que se intentará buscar un recubrimiento mínimo para grafos y árboles.

Estos problemas se van a resolver mediante algoritmos greedy (voraz), los cuales se basan en obtener una solución lo más óptima posible en función de la elección local de la opción más óptima, en el caso del vecino más cercano, sería escoger la ciudad que menos distancia tenga desde donde esté el viajante.

Así, se va a seguir una serie de pasos en los que identificamos las características de cada algoritmo, que son:

Conjunto candidatos: Objetos susceptibles a ser elegidos como parte de la solución
Conjunto seleccionados: Objetos elegidos como parte de la solución
Función solucion: Criterio que indica cuando un conjunto de candidatos es solución
Función selección: Criterio que indica cuando un conjunto de candidatos podría llegar a ser una solución
Función factibilidad: Criterio que indica que candidato no usado es el mejor en cada momento
Función objetivo: Criterio que establece la solución, puede ser el mismo que la selección.

2. Hardware y Software

La compilacion y ejecucion de los algoritmos desarrollados en esta práctica se han realizado en un sistema linux con la distribución “Arch Linux” de 64 bits, con un procesador i7-4510U a 2GHz.

```
$uname -rms
Linux 5.5.8-arch1-1 x86_64
```

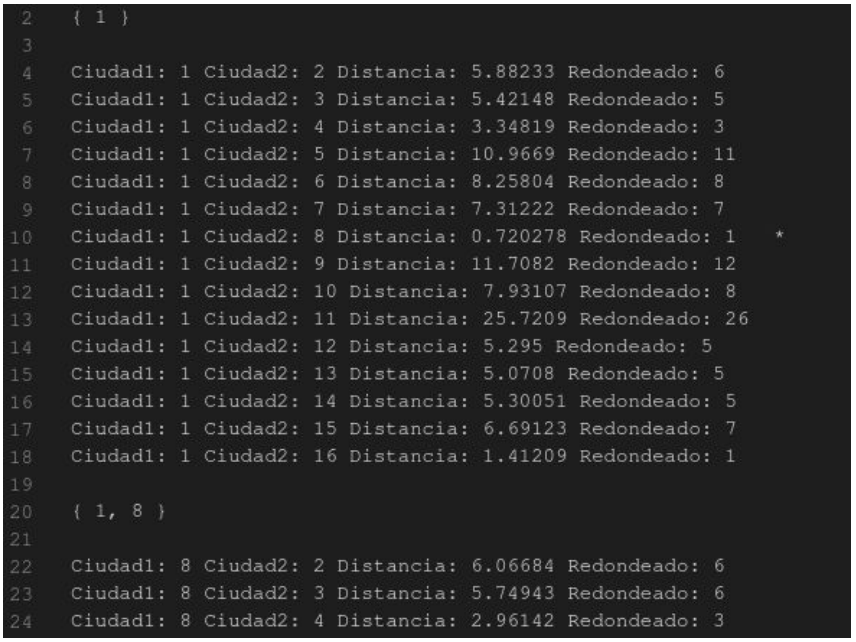
3. Travelling Salesman Problem

a. Cercanía

En este apartado vamos a usar la heurística del vecino más cercano, esta heurística consiste en visitar la ciudad más cercana a la ciudad en la que esté el viajante.

Así, para cada ciudad del circuito, se calcula una ruta hasta recorrerlas todas incluyendo la vuelta a la ciudad origen, quedándose con la ruta que menos distancia tenga.

En la imagen siguiente se puede ver el proceso de elección de ciudad, el viajante está en la ciudad 1 y se comprueba la distancia a todas las ciudades restantes {2 → 16} eligiendo la 8 por ser la que menos distancia tiene. Una vez que ha elegido la ciudad más cercana, repite el proceso a partir de la ciudad anterior (8).



Características Greedy

Conjunto candidatos	Todas las ciudades
Conjunto seleccionados	Ciudades no visitadas
Función solución	Ruta mínima entre ciudades
Función selección	Ciudad no visitada más cercana al viajero
Función factibilidad	Ruta es mínima (variando ciudad origen)
Función objetivo	Elegir ciudad origen mediante la cual pase por todas las ciudades con una distancia mínima

Pseudocódigo

1 - Inicializar solución con la ruta de ciudades leídas de fichero incluyendo la ciudad retorno
2 - Calcular distancia de la solución inicial
3 - Para cada ciudad del circuito
4 - Iniciar vector visitados con la ciudad origen
5 - Iniciar candidatos con las ciudades no visitadas
6 - Mientras haya ciudades candidatas
7 - Obtener posición de la ciudad más cercana
8 - Añadir ciudad elegida al vector de visitados
9 - Eliminar ciudad visitada del vector candidatos
10 - Añadir ciudad origen
11 - Calcular distancia ruta de visitadas
12 - Si la distancia de la ruta de visitadas mejora respecto a la solución anterior
13 - Actualizar solución con la ruta de visitadas
14 - Intercambiar ciudad origen por una ciudad por la que no se haya empezado

Para este problema hemos usado una estructura (struct) en la que se almacena el identificador y las coordenadas de cada ciudad. Para almacenar las ciudades candidatas a solución se ha usado un vector de ciudades, que va almacenando cada ciudad candidata, al igual que se usa un vector de ciudades para almacenar la ruta solución.

Distancias y Tiempos

A continuación se puede ver una tabla de tiempos y la distancia de la ruta obtenida para la heurística de cercanía.

	Tiempo	Distancia	Distancia Óptima
Ulysses16	0,000324741	110	73
Ulysses22	0,000712269	124	74
att48	0,0067376	40113	33522
a280	1,05573	3105	2579

Como se puede ver, el algoritmo implementado no obtiene la distancia óptima (calculada a partir de los ficheros opt.tour), esta conclusión es lógica porque puede ser que al viajante le convenga ir primero a una ciudad que está más lejos que a otra que está más cerca, ya que no se mira por la solución final sino por el momento en el que está el viajante.

La eficiencia del algoritmo por el vecino más cercano es de $O(n^2)$, ya que para cada ciudad no visitada tiene que buscar entre todas esas ciudades cuál es la más cercana (bucle “for” dentro de un “while”), pero como además, se ha implementado para que calcule una ruta para cada ciudad del circuito (puede ser que empezando por la ciudad 3, la distancia de la ruta sea mejor que empezando desde la ciudad 2), se convierte en un algoritmo de eficiencia $O(n^3)$

(Haciendo referencia al pseudocódigo)

$O(n^3)$ → “for” (linea 3) que contiene un “while” (linea 6) que contiene a su vez un “for” (linea 7)

b. Inserción

Esta heurística de inserción consiste en buscar la posición en la que el aumento de la distancia de la ruta es mínima, es decir, se elige la ciudad cuyo coste de inserción entre dos ciudades es el más pequeño. Para ello, se parte de una solución inicial con las ciudades más al Norte, más al Este y más al Sur.

Para almacenar la ruta se usa un vector de pares de ciudades, donde el primer elemento indica la ciudad desde donde sale y el segundo elemento la ciudad a la que va. La ventaja de usar esta estructura, es que no se necesita insertar en orden las ciudades, ya que para cada par sabemos el destino de cada ciudad. Cada par del vector representa una arista de la ruta.

Características Greedy

Conjunto candidatos	Todas las ciudades
Conjunto seleccionados	Ciudades no visitadas
Función solución	Ruta mínima entre ciudades
Función selección	Ciudad no visitada que minimice el aumento de distancia entre dos ciudades
Función factibilidad	Aumento de la distancia entre dos ciudades al insertar es mínimo
Función objetivo	Ciudad no visitada que minimice el aumento de distancia entre dos ciudades

Pseudocódigo

```
1 - Crea vector de pares de ciudades
2 - Buscar posiciones de las ciudades más al norte, este y sur.
3 - Añade el par [Norte, Este], el par [Este, Sur] y el par [Sur, Norte]
4 - Rellena el vector de candidatas con las ciudades excluyendo las ciudades Norte, Este y Sur
5 - Mientras haya ciudades candidatas
6 -     Para cada ciudad candidata
7 -         Para cada posición del vector de visitadas
8 -             Calcular el aumento de la distancia donde se inserta la ciudad candidata
9 -             Si mejora la distancia al insertar
10 -                Actualizar posición de la ciudad a insertar
11 -                Actualizar posición donde se inserta la ciudad candidata
12 -    Guardar la segunda posición del par a modificar (destino de la arista)
13 -    Añadir la ciudad elegida en la posición a insertar en el par de ciudades (inicio de la arista)
14 -    Añadir nuevo par con la ciudad a añadir y la modificada (añadir la arista)
15 -    Eliminar ciudad añadida de candidatos
16 - Ordenar vector de visitadas y actualizar solucion
```

Distancias y Tiempos

A continuación se puede ver una tabla de tiempos y la distancia de la ruta obtenida para la heurística de inserción:

	Tiempo	Distancia	Distancia Óptima
Ulysses16	0,000180596	119	73
Ulysses22	0,000489116	84	74
att48	0,00467251	37474	33522
a280	0,870394	3115	2579

Como se puede ver, al igual que con la heurística del vecino más cercano, tampoco se obtiene la ruta óptima para este problema, aunque las distancias han mejorado respecto al punto anterior.

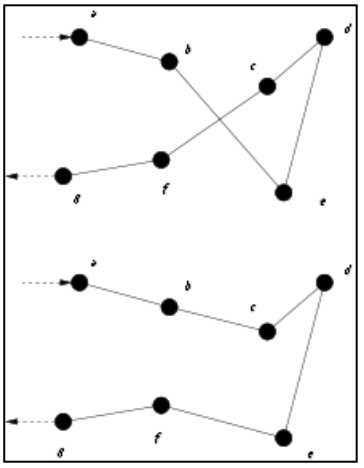
No se obtiene la ruta óptima ya que el criterio usado es elegir la ciudad candidata cuyo incremento en la distancia de la ruta sea el mínimo, y puede darse el caso en el que dicha ciudad sea mejor insertarla en una posición donde aumente más la distancia, e insertar otra ciudad en la posición donde se iba a colocar, mejorando aún más la distancia.

(Haciendo referencia al pseudocódigo)

La eficiencia de este algoritmo usando la heurística de Inserción es de $O(n^3)$, ya que mientras queden ciudades candidatas por insertar en la ruta (línea 5 “while” de O(n)), vamos comprobando una a una entre qué pares de ciudades es más óptimo insertarlas (línea 6 y 7, “for” y “for” O(n²)), consiguiendo así un aumento mínimo de la distancia total de la ruta.

c. Nueva estrategia: 2-opt

Esta heurística se basa en el intercambio de un tramo de una ruta, buscando minimizar la distancia de la ruta; hablando coloquialmente se podría decir que busca “deshacer cruces”, ya que un cruce implica pasar dos veces por el mismo punto.



En la imagen de la derecha se puede observar un ejemplo:

Ruta inicial: A → B → E → D → C → F → G → A
Realiza el intercambio:

1. Añade una parte desde el inicio de la ruta
    a. **(A → B)**
2. Añade una parte intermedia en orden inverso
    a. A → B → **(C → D → E)**
3. Añade el resto de la ruta
    a. A → B → (C → D → E) → **(F → G → A)**

Así, lo que hace greedy a esta heurística es que se busca una ruta que mejore la distancia de la ruta actual sin intercambiar, de esta forma, cuando termina de comprobar para una ciudad de la ruta las posibles combinaciones de intercambio y ha encontrado una mejora en la ruta, deja de comparar y pasa a comprobar la ruta para la siguiente ciudad de la ruta.

En este problema se usan las mismas estructuras que en la heurística del vecino más cercano, un struct que almacena el identificador de la ciudad y sus coordenadas, un vector de ciudades para almacenar la ruta candidata de ciudades y un vector solución de ciudades.

Carasteristicas Greedy

Conjunto candidatos	Todas las ciudades
Conjunto seleccionados	Ciudades que mejoran la ruta inicial
Función solución	Ruta sin cruce
Función selección	Distancia de la ruta con el mejor tramo de ciudades intercambiadas
Función factibilidad	Distancia ruta es mínima (no hay cruces)
Función objetivo	Distancia de la ruta con el mejor tramo de ciudades intercambiadas

Pseudocódigo

```
1 - Inicializar solución mediante el algoritmo del vecino más cercano o sobre un fichero leído
2 - Mientras haya mejora en la ruta
3 -     Establecer que no hay mejora
4 -     Calcular la distancia de la ruta
5 -     Para cada ciudad de la ruta “i”, excluyendo origen y retorno, y si no hay mejora
6 -         Para cada posible intercambio desde “j=i+1” hasta el “final-1”
7 -             Realizar intercambio desde “j a i”
8 -             Calcular distancia de la ruta intercambiada
9 -             Si mejora la distancia
10 -                Actualizar solucion
11 -                Establecer que hay mejora
```

Distancias y Tiempos

A continuación se puede ver una tabla en la que se reflejan tiempos de ejecución y distancias de las rutas obtenidas mediante este algoritmo, partiendo desde una ruta base (leída del fichero .tsp) y una ruta solución greedy generada mediante la heurística del vecino más cercano (VMC)

	Sin solución Greedy previa		Con solución Greedy previa (VMC)		Distancia Óptima
	Tiempo	Distancia	Tiempo	Distancia	
Ulysses16	0,00423473	116	0,00270537	107	73
Ulysses22	0,00815656	118	0,0036973	112	74
att48	0,206673	45378	0,0696617	37535	33522
a280	17,0613	3077	46,9765	2772	2579

Como se puede observar, los tiempos de ejecución partiendo de una solución previa son mejores que generando una solución desde 0. Esto se debe a que partiendo de una solución previa, el número de iteraciones a realizar en dicha ruta es mucho menor que si tiene que generar la solución desde 0, pero también se puede observar que para el fichero “a280” el tiempo es mucho mayor que los demás.

En el caso del fichero “a280”, la distancia obtenida mediante el vecino más cercano es de “3105”, como se puede ver, este algoritmo mejora esa distancia en un tiempo relativamente bajo hasta “3077”. Sin embargo, si se realiza el algoritmo a partir de una solución previa, se puede ver que la distancia mejora muchisimo mas, hasta “2772” a coste del tiempo. Esto se debe a que aunque generalmente, empezar desde una solución previa implica reducir el número de iteraciones del algoritmo, si dicha solución es “mala” o no es una solución “buena” (no se acerca sustancialmente a la óptima), dicho número de iteraciones no es tan pequeño, por lo que pierde eficiencia.

Antes de explicar la eficiencia del algoritmo, tenemos que decir que el peor caso se produce cuando siempre se está haciendo mejora en la ruta, ya que la condición de salida es “repetir mientras haya mejora”, lo que implica que a más tiempo tarde el algoritmo, mejor resultado dará.

(Haciendo referencia al pseudocódigo)

La eficiencia de este algoritmo es $O(n^4)$ ya que en el peor caso, siempre se está realizando mejora (O(n)) (línea 2), así, cada vez que se encuentra una mejora, repite el proceso para cada ciudad de la ruta (O(n)) (línea 5) en la que realiza los (n-1) posibles intercambios ($O(n^2)$) (línea 6 y 7),

d. Ficheros de pruebas

Para la ejecución de estos algoritmos se han usado ficheros “.tsp” y “tour.tsp” proporcionados por el profesorado, dichos archivos contienen una cabecera de información, de la cual se extrae la cantidad de ciudades en la ruta. A continuación se leen las ciudades y sus coordenadas.

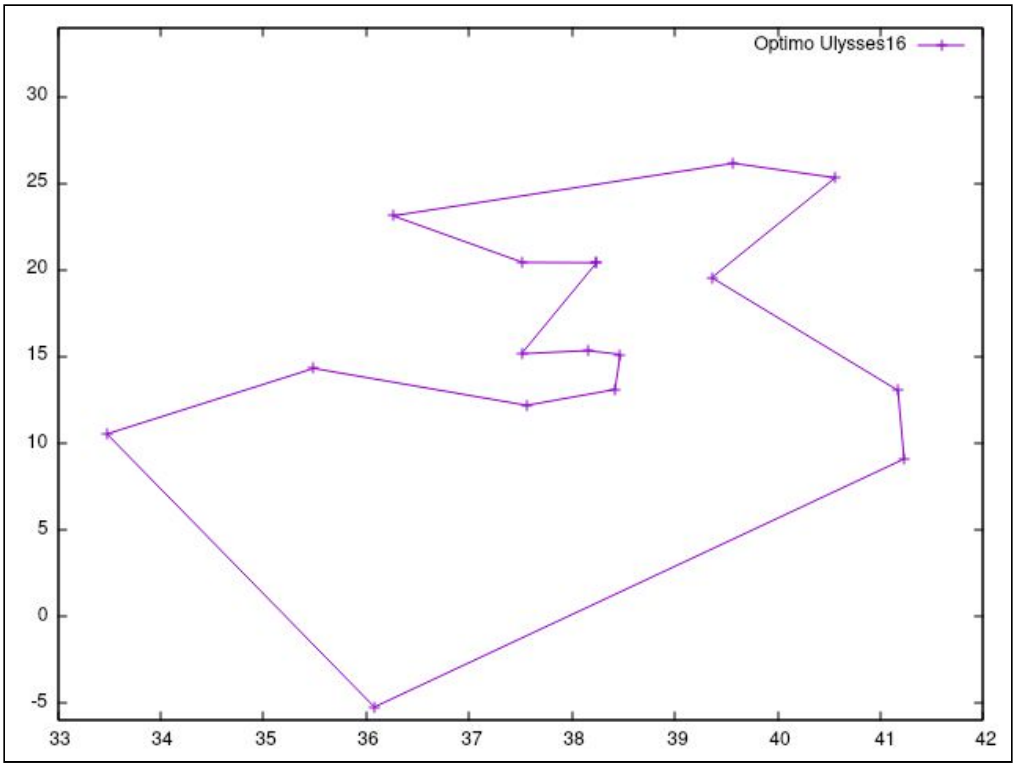
e. Comparativa

A continuación se van a mostrar las distintas resoluciones de nuestros algoritmos para una misma ruta base, empezado por la ruta óptima para ese circuito proporcionada en un fichero con extensión “opt.tour” y luego mostrando cada ruta obtenida por nuestros algoritmos.

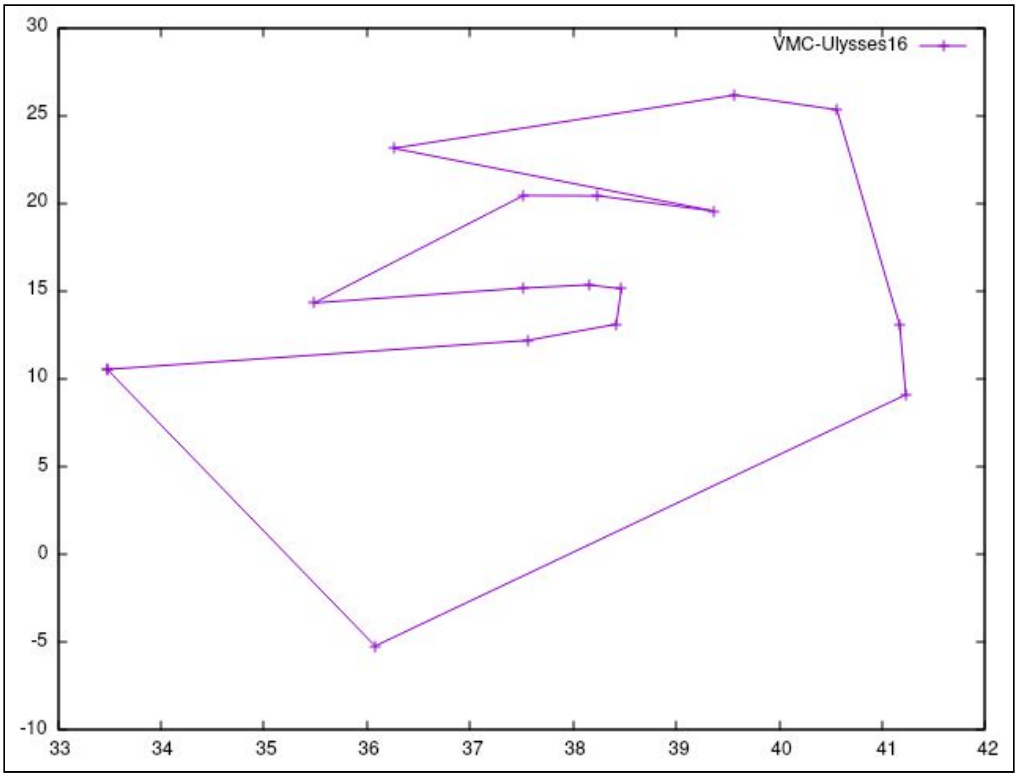
Como se va a ver, la ruta obtenida por la heurística del vecino más cercano no se asemeja a la ruta óptima, mientras que por inserción la ruta empieza a obtener una pequeña similitud, con 2-opt sí obtiene una aproximación bastante buena en el caso de usar una solución previa Greedy por el vecino más cercano, mientras que partiendo de una ruta base (igual que con cercanía e inserción), la ruta obtenida no es tan cercana a la óptima.

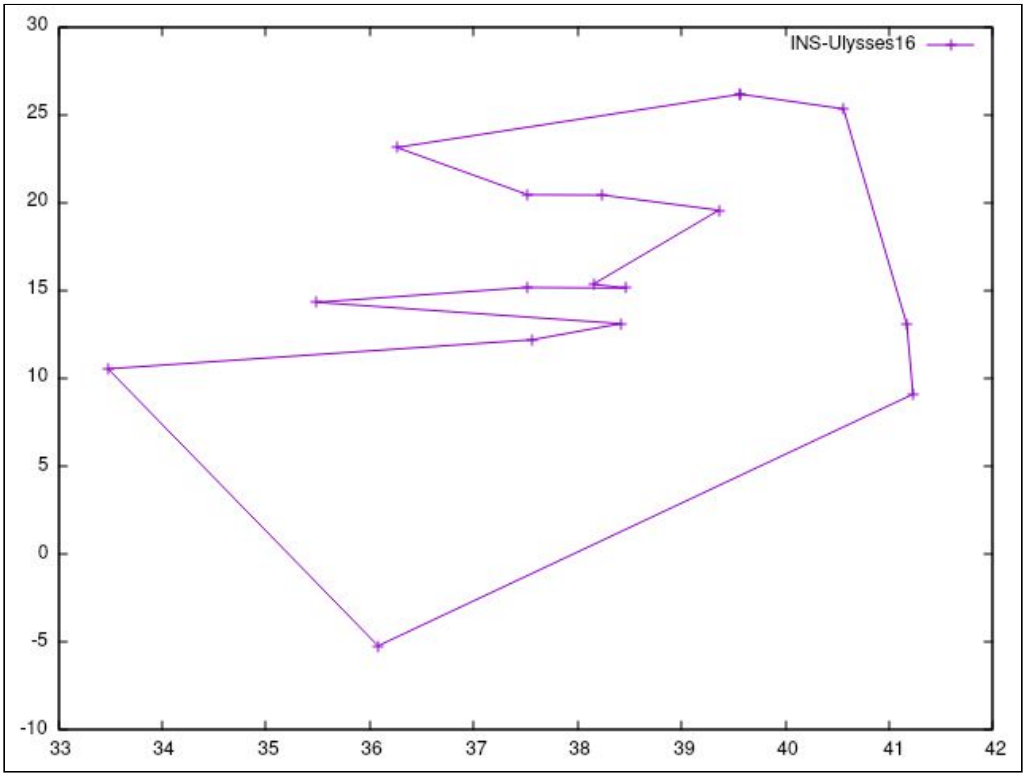
Ulysses16

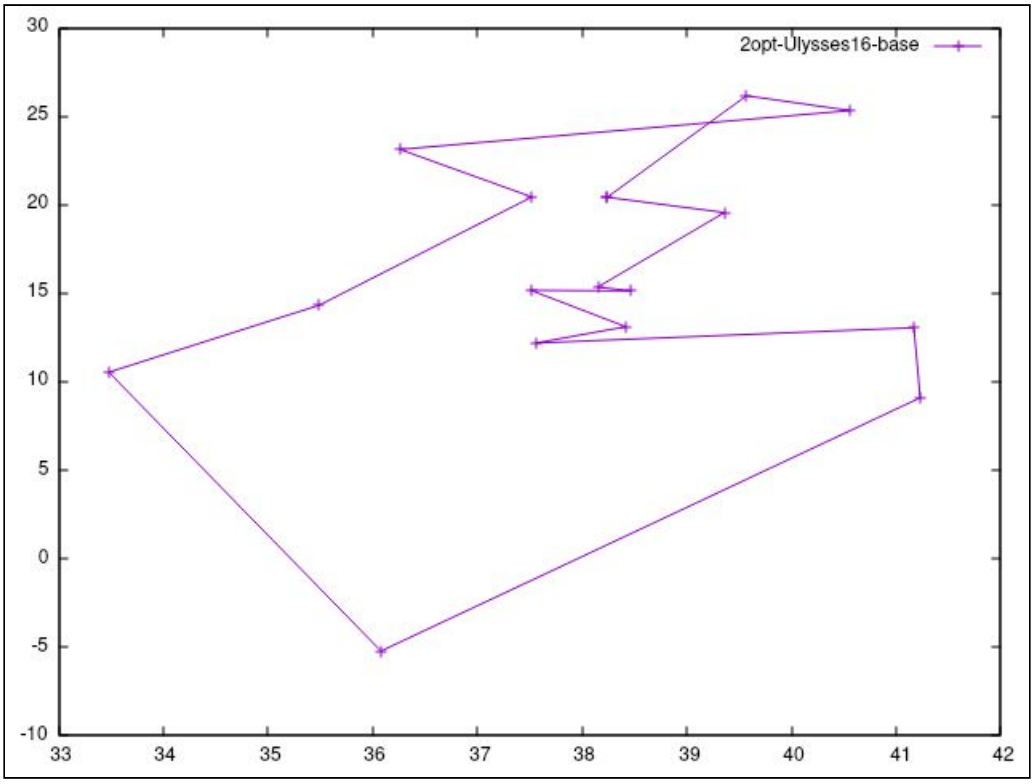
La ruta óptima es la que se ve a continuación:

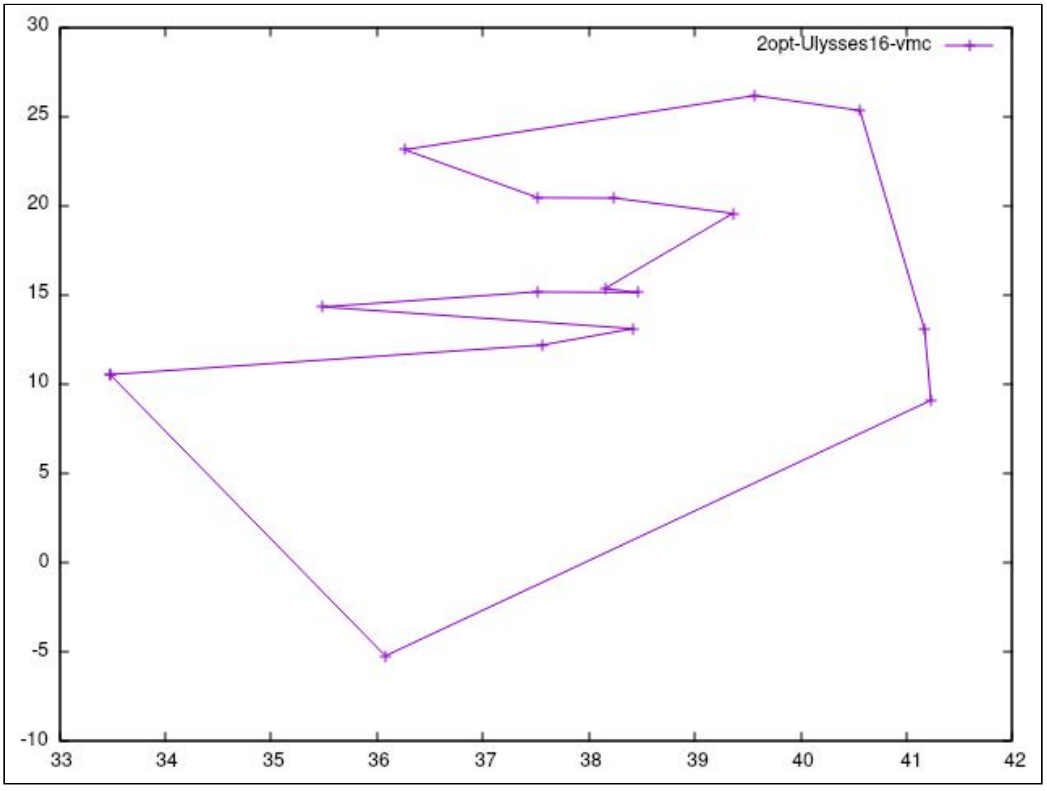


Las siguientes imágenes son las rutas obtenidas mediante el vecino más cercano, inserción y 2-opt.



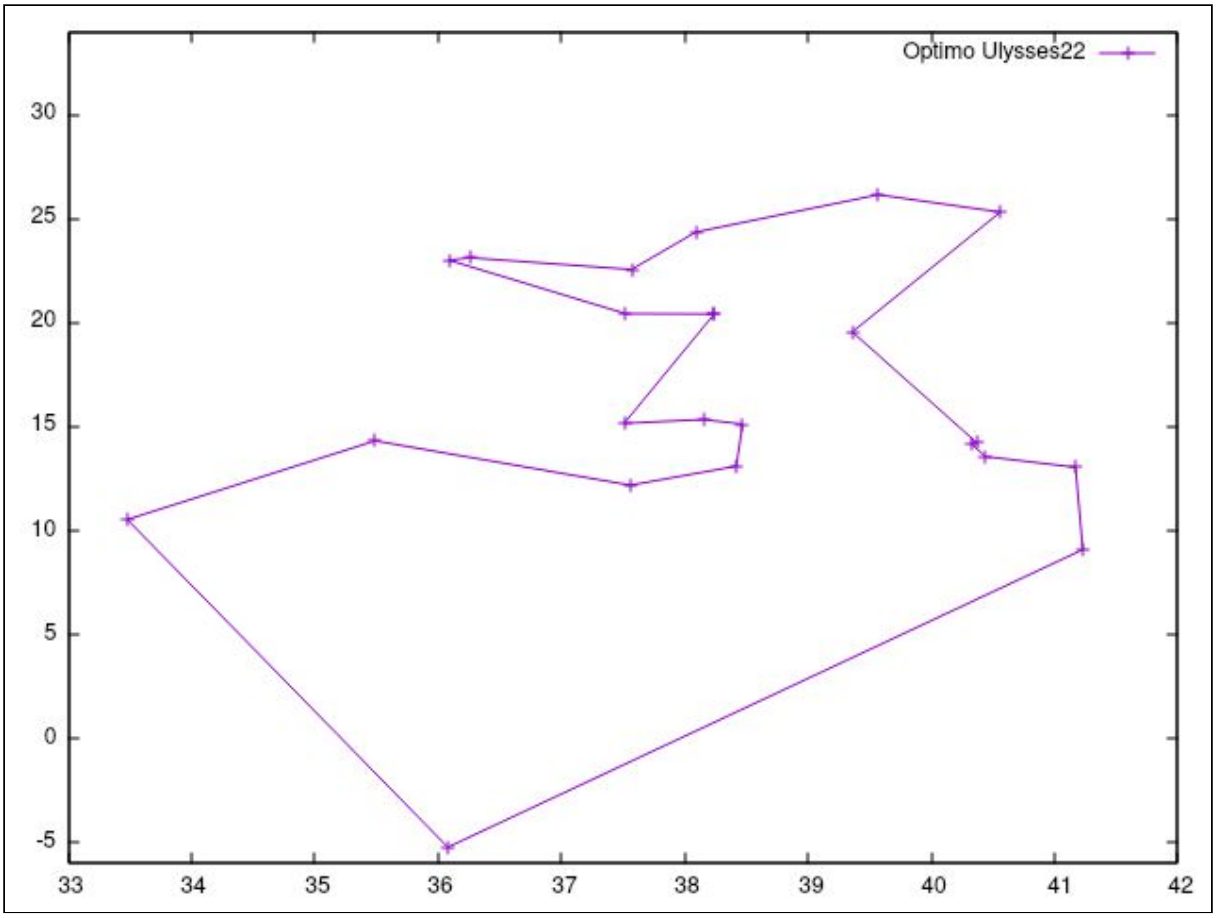



2opt-Ulysses16-base
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42

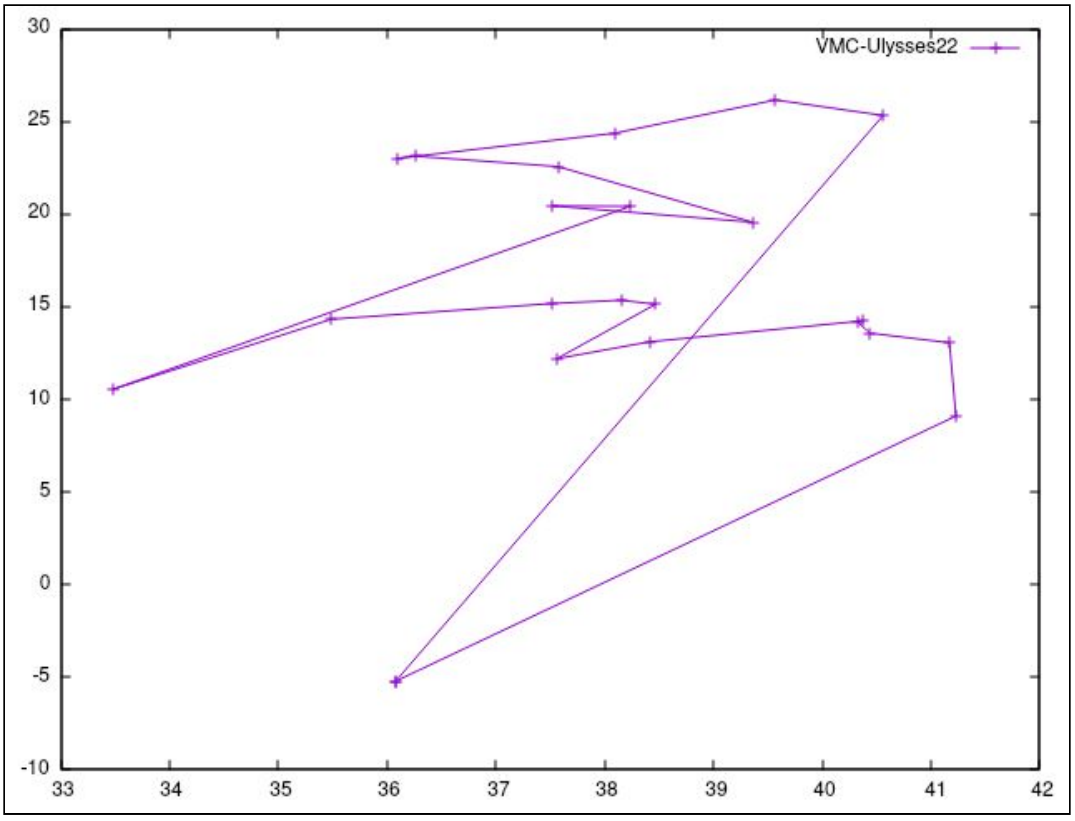

2opt-Ulysses16-vmc
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42

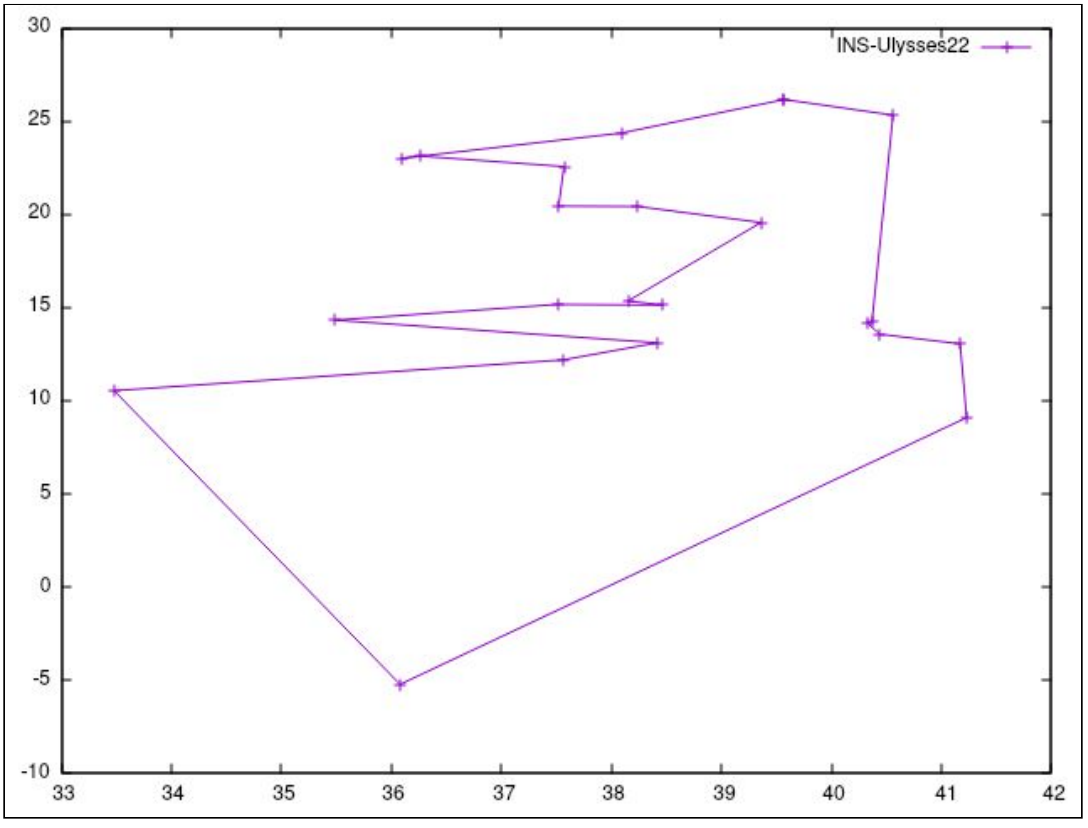
Ulysses22

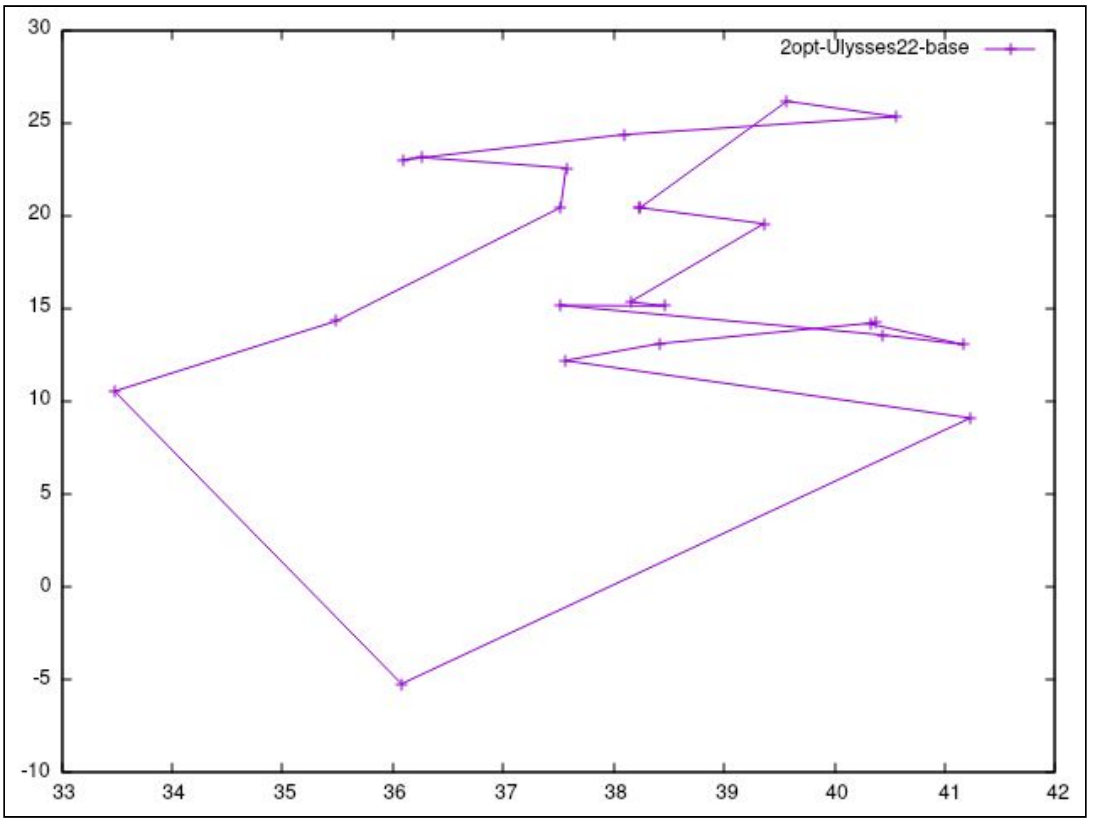
La ruta óptima es la que se va continuación:

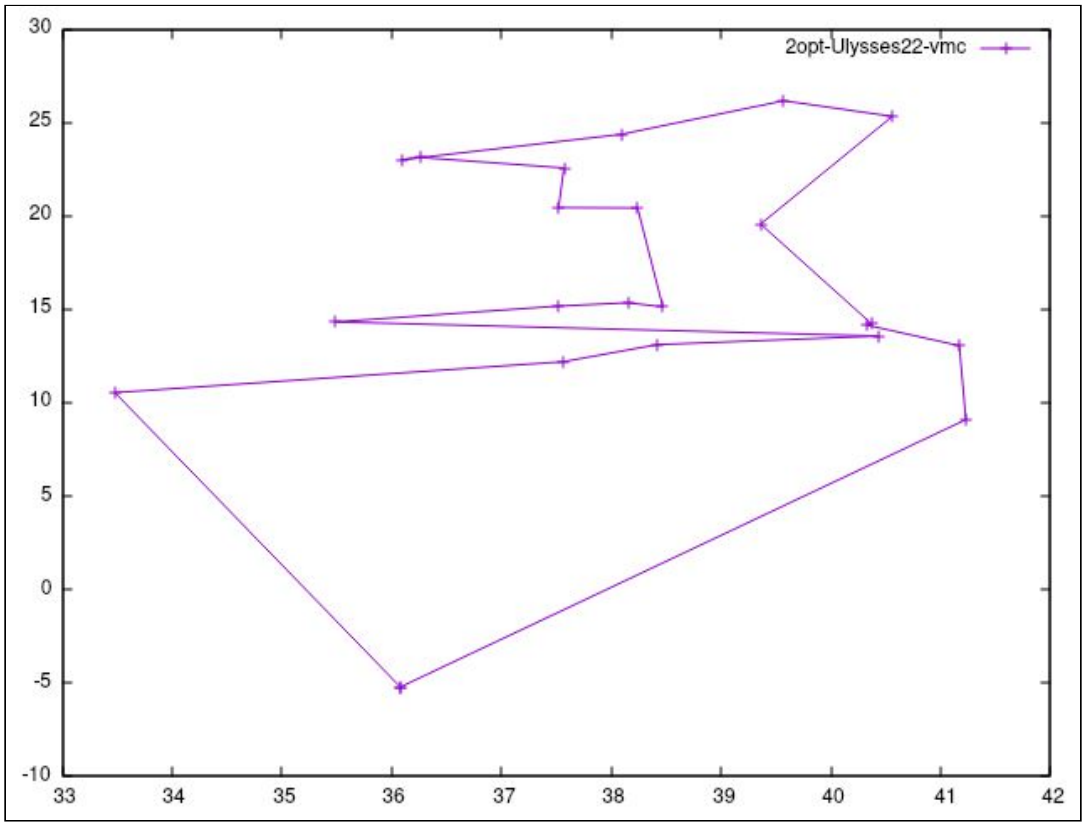


Las siguientes imágenes son las rutas obtenidas mediante el vecino más cercano, inserción y 2-opt.



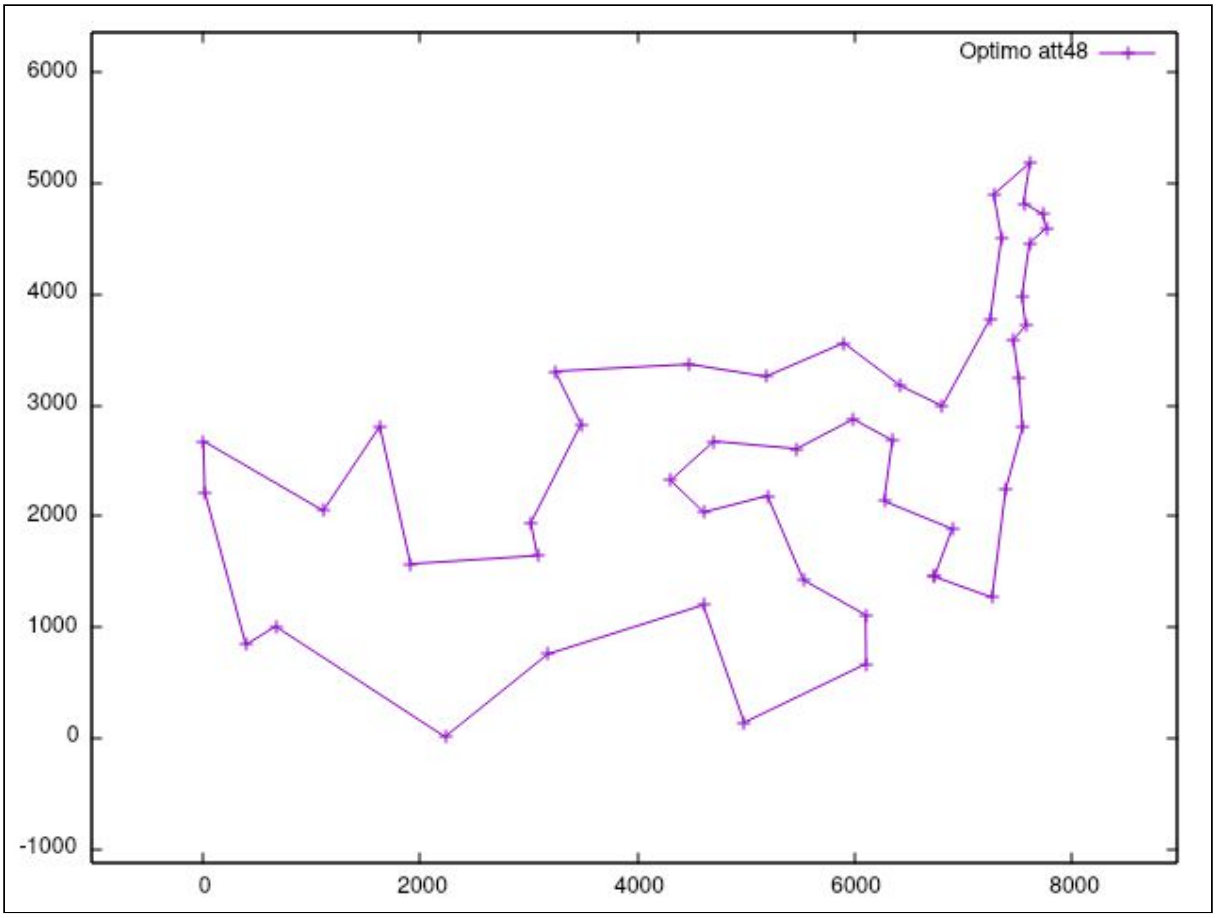



2opt-Ulysses22-base
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42

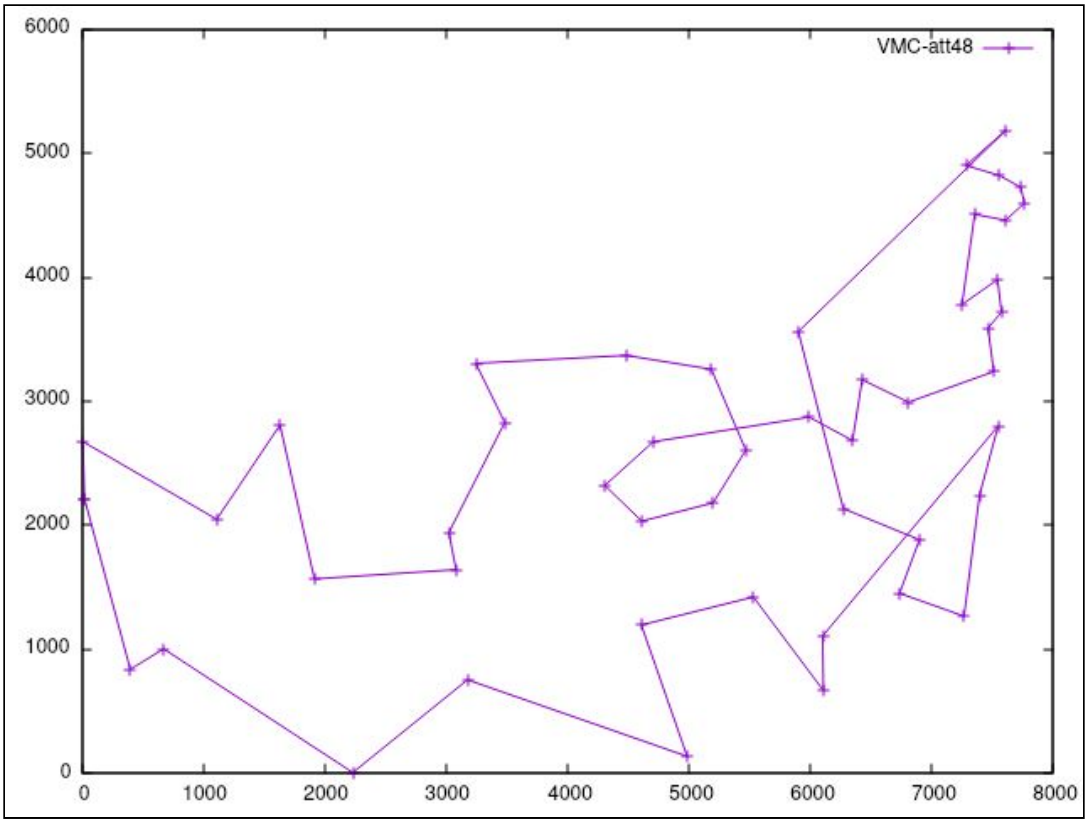

2opt-Ulysses22-vmc
30
25
20
15
10
5
0
-5
-10
33
34
35
36
37
38
39
40
41
42

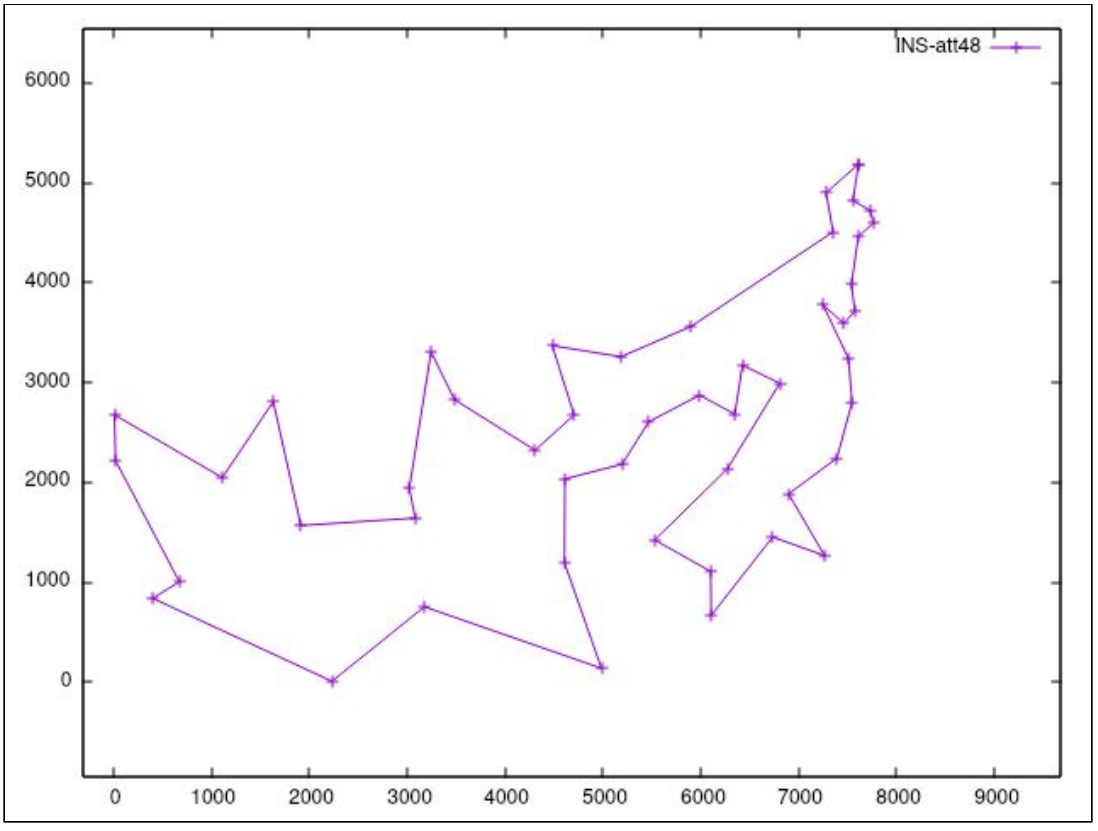
att48

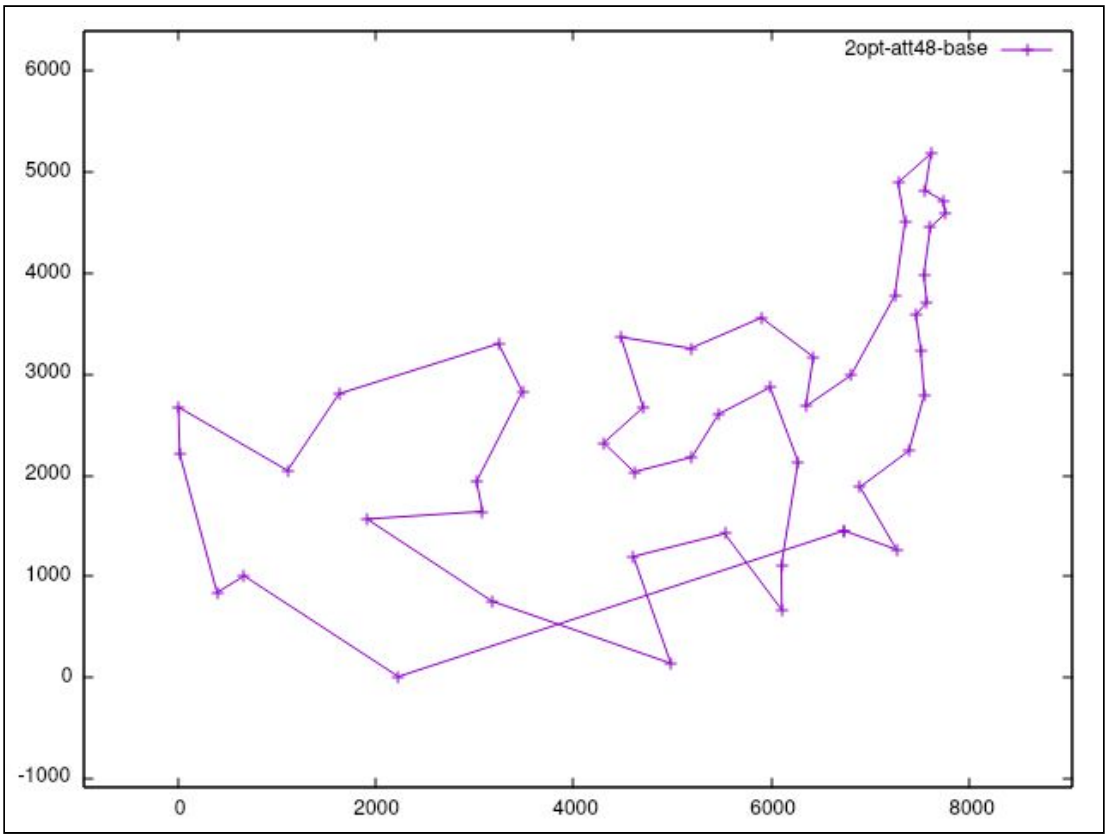
La ruta óptima es la que se va continuación:

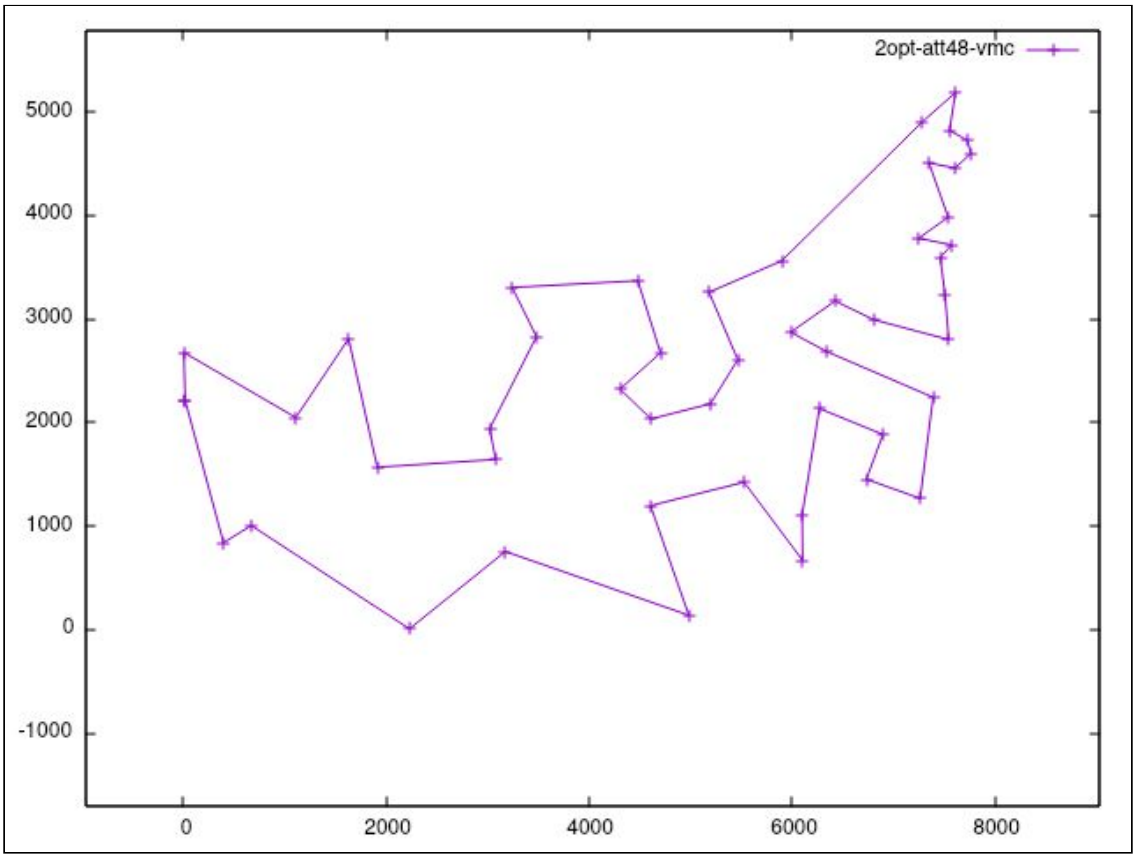


Las siguientes imágenes son las rutas obtenidas mediante el vecino más cercano, inserción y 2-opt.



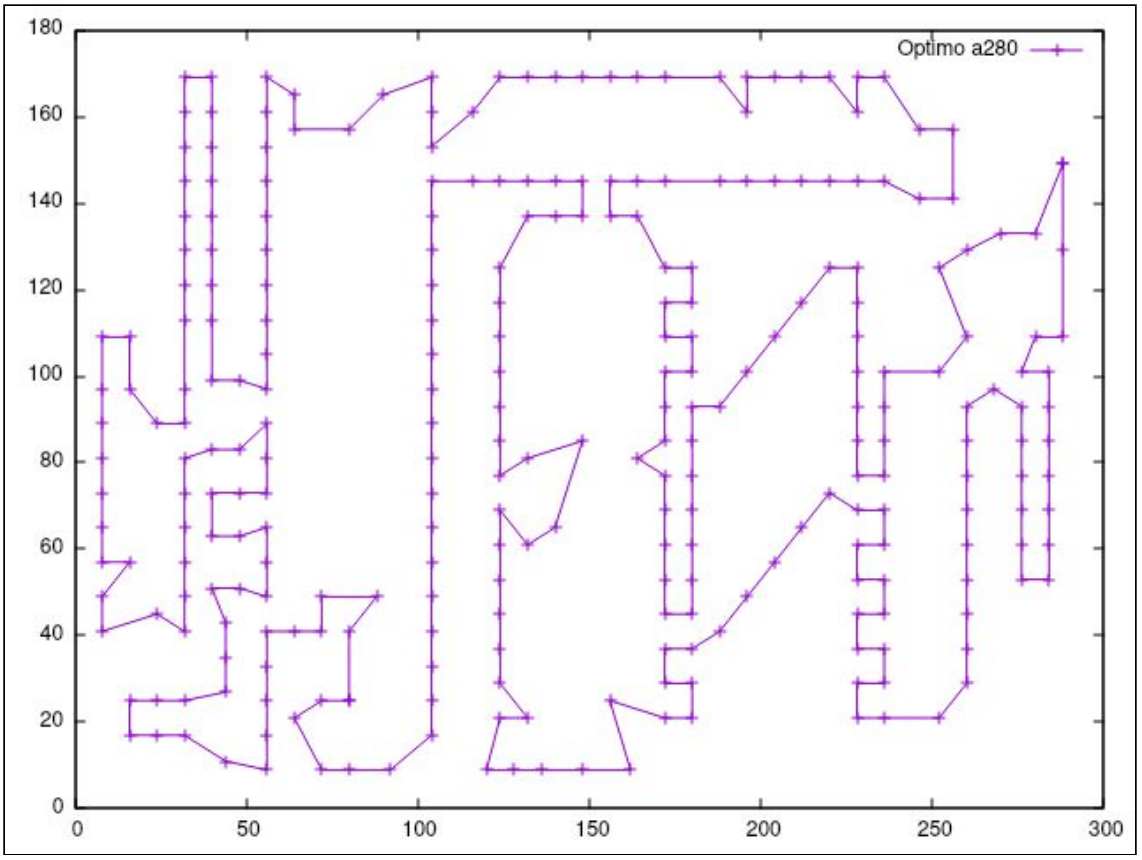



2opt-att48-base
6000
5000
4000
3000
2000
1000
0
-1000
0
2000
4000
6000
8000

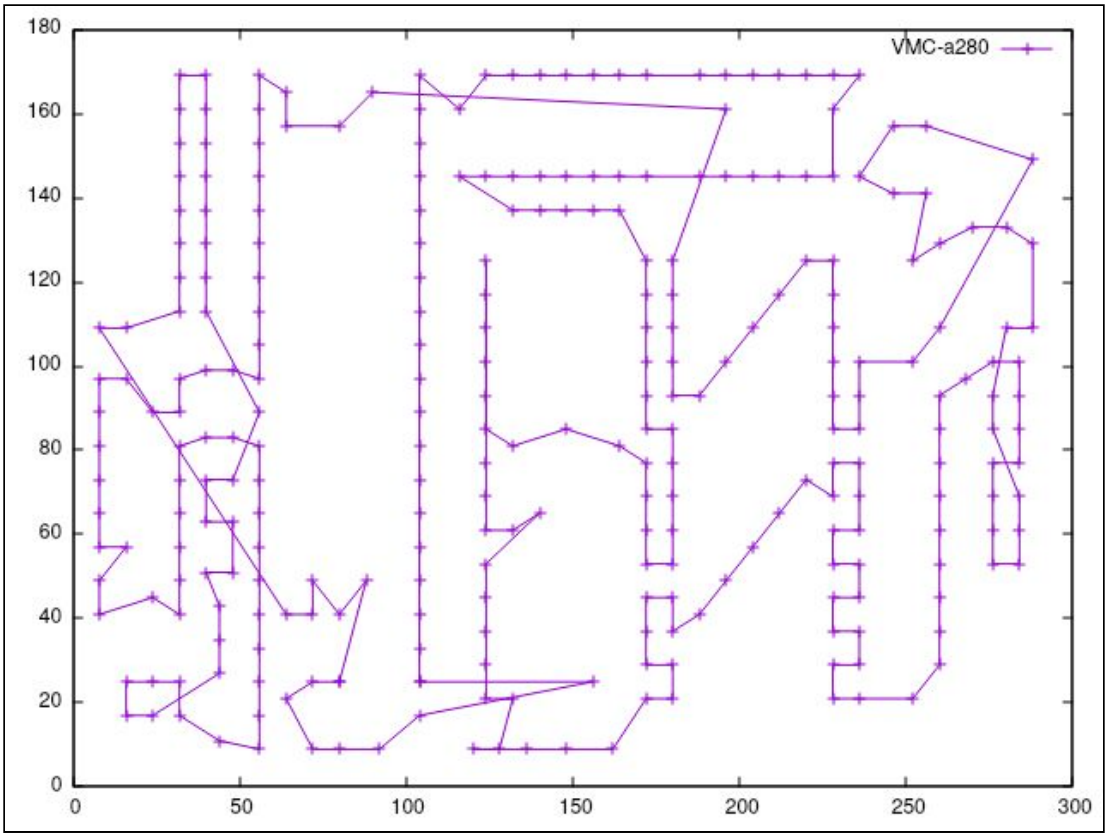

2opt-att48-vmc
5000
4000
3000
2000
1000
0
-1000
0
2000
4000
6000
8000

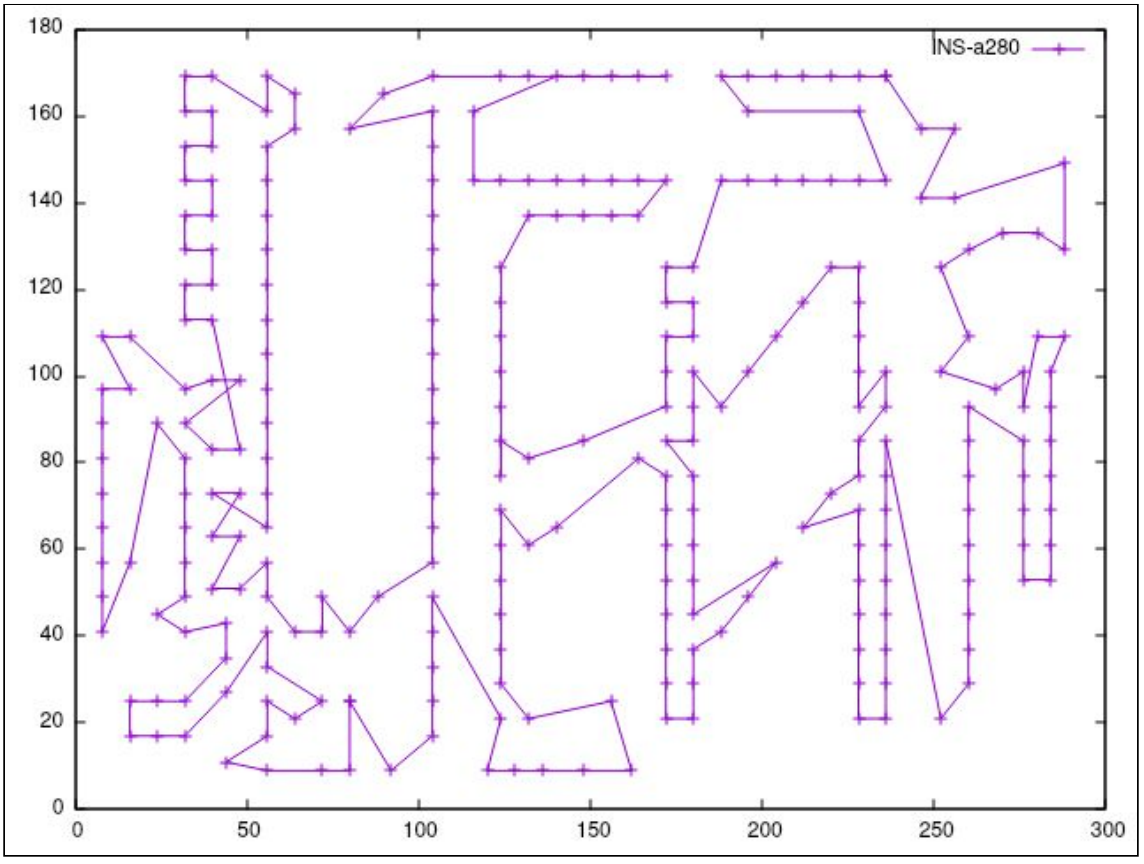
a280

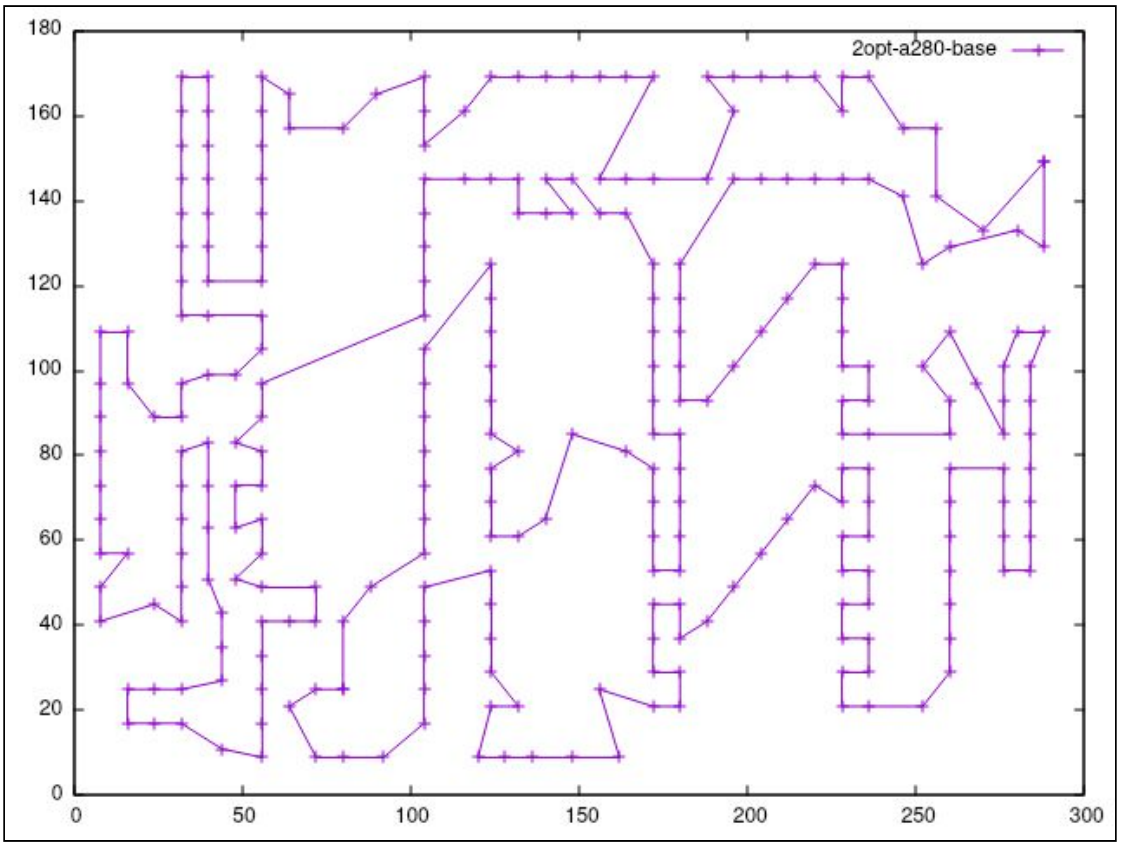
La ruta óptima es la que se va continuación:

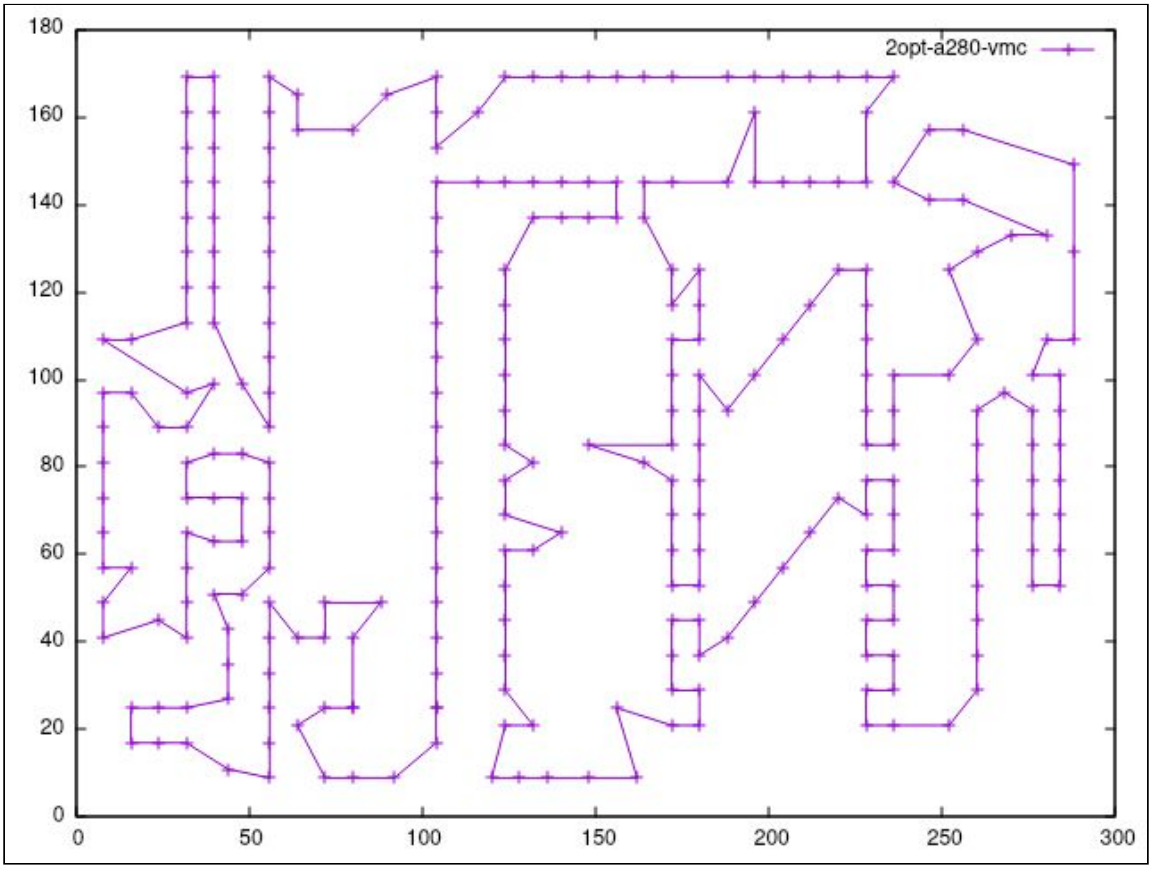


Las siguientes imágenes son las rutas obtenidas mediante el vecino más cercano, inserción y 2-opt.






2opt-a280-base
180
160
140
120
100
80
60
40
20
0
0
50
100
150
200
250
300


2opt-a280-vmc
180
160
140
120
100
80
60
40
20
0
0
50
100
150
200
250
300

4. Recubrimiento de un grafo no dirigido

El recubrimiento de un grafo no dirigido, consiste en localizar aquellos nodos que cubran todas las aristas del grafo. En este ejercicio se busca que dichos nodos sean el mínimo posible, de forma que se obtenga el recubrimiento minimal.

Se van a implementar dos versiones, una para grafos generales y otra para el caso específico en el que un grafo sea un árbol.

Las estructuras usadas en este problemas son las siguientes:

- Map con clave entera y valor de vector de enteros
    - Almacena la estructura del grafo/árbol, donde la clave es un nodo y en el vector asociado se almacenan sus conexiones.
- Matriz de adyacencia:
    - Matriz que establece entre qué nodos hay una arista.
    - Es una matriz simétrica, la diagonal indica si hay un bucle en el nodo.
    - La suma de cada fila/columna, indica el grado de dicho nodo.
    - Un 1 indica que hay conexión, un 0 indica que no hay conexión
- Vector de enteros “nodos”
    - Cada posición del vector representa un nodo y su valor, el grado de dicho nodo.
- Vector de enteros “solución”
    - Almacena los nodos elegidos solución del grafo/árbol

a. Grafo

En este apartado se ha implementado un algoritmo Greedy para buscar el recubrimiento minimal de un grafo, dicho algoritmo se basa en buscar el nodo del grafo con más grado (mayor número de aristas conectadas a él) y tomarlo como solución, reduciendo el grado de los nodos conectados a él en 1 (sus nodos adyacentes pierden la arista conectada)

Características Greedy

Conjunto candidatos	Todos los nodos
Conjunto seleccionados	Nodos de mayor grado
Función solución	Nodos que recubren minimalmente el grafo
Función selección	Nodo con mayor grado
Función factibilidad	No hay más nodos con grado > 0
Función objetivo	Nodo con mayor grado

Pseudocódigo

1 - Almacena los datos recibidos en la matriz de adyacencia del grafo
2 - Se crea un vector que almacena el grado de cada uno de los nodos
3 - Se comprueba la diagonal de la matriz de adyacencia
4 - Se añaden a la solución los nodos que estén unidos a sí mismos
5 - Se establece el grado de estos a cero y se reduce en uno el grado de los nodos adyacentes
6 - Mientras queden nodos con grado mayor que cero
7 - Se busca el nodo que tiene el mayor grado actualmente
8 - Se añade este nodo a la solución y se establece su grado a cero
9 - Se reduce en uno el grado de los nodos adyacentes

(Haciendo referencia al pseudocódigo)

La eficiencia en este algoritmo viene marcada por el número de grados (aristas) del grafo más que por el número de nodos, ya que para un grado con 100 nodos donde 99 de ellos están conectados a 1 solo, se tendría un nodo de grado 99 y 99 nodos de grado 1, por lo que el “while” de la línea 6 se ejecutaría 1 sola vez y el “for” de la línea 7 que busca el nodo de mayor grado, en el peor del mejor caso recorre todos los nodos (O(n)), el cual marcaría solución dicho nodo terminando el algoritmo con una eficiencia O(n).

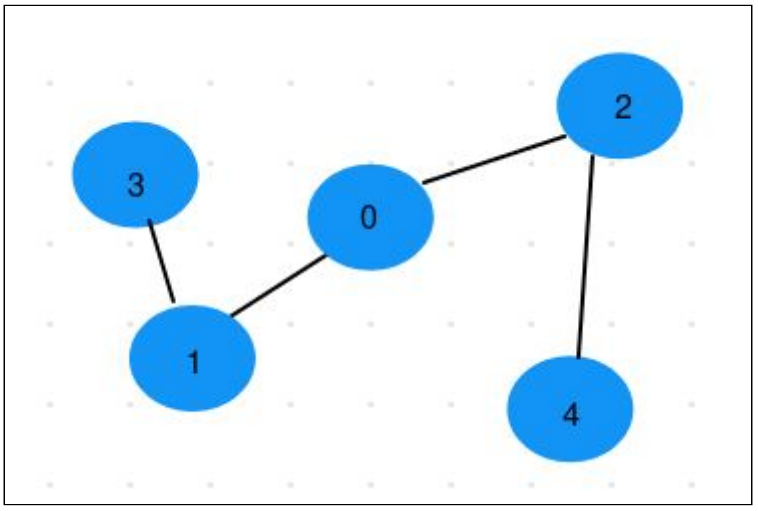
En cambio, si hay muchos nodos y todos ellos tienen grados muy altos, se produciría el peor caso posible, ya que el “for” que busca el nodo de mayor grado recorrería todo el vector de nodos (O(n)) y al reducir el grado de dicho nodo y sus adyacentes.

En el peor de los peores casos, el vector de búsqueda del nodo con mayor grado siempre se recorre hasta el final, escogiendo el último nodo del vector (O(n)) y al reducir el grado de los nodos adyacentes, reduzca solamente el grado de un nodo, de forma que la reducción de grados es mínima, produciendo, que el número de nodos con grado mayor a 0, sea siempre el máximo posible, de esta forma se obtendría una eficiencia $O(n^2)$

i. Demostración de no minimalidad en grafos

Al aplicar nuestro algoritmo a un grafo, este proporciona un recubrimiento que en principio parece minimal, pero en determinados casos, no nos asegura que este sea un recubrimiento minimal.

En el siguiente contraejemplo lo podemos comprobar:



Empezamos buscando el nodo de mayor grado, que en este caso tenemos tres (1, 2 y 3) con grado 2 .

Vector solución	{Vacío}
Vector candidatos	{2, 2, 2, 1, 1}

Con una matriz de adyacencia:

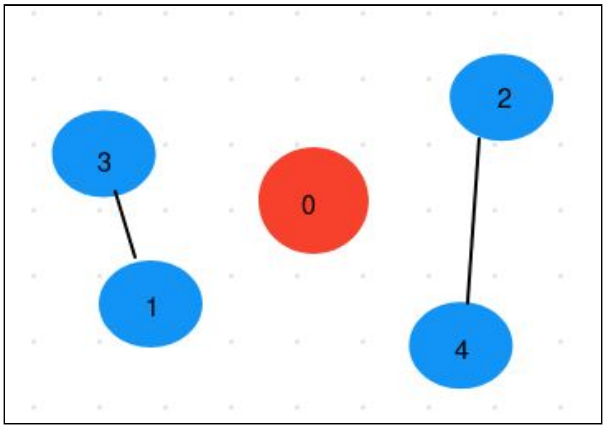
Nodo	0	1	2	3	4	Grado
0	0	1	1	0	0	2
1	1	0	0	1	0	2
2	1	0	0	0	1	2
3	0	1	0	0	0	1
4	0	0	1	0	0	1

En azul se representan los nodos, en rojo claro se representa el vector de candidatos, donde la posición del vector representa el nodo y el valor el grado del nodo, las casillas verdes indican una conexión entre dos nodos.

El siguiente paso es buscar en el vector de candidatos, el nodo (posición) con mayor grado, que sería el nodo 0 que está en la posición 0 (aunque haya nodos posteriores con el mismo grado se queda con el primero que encuentra).

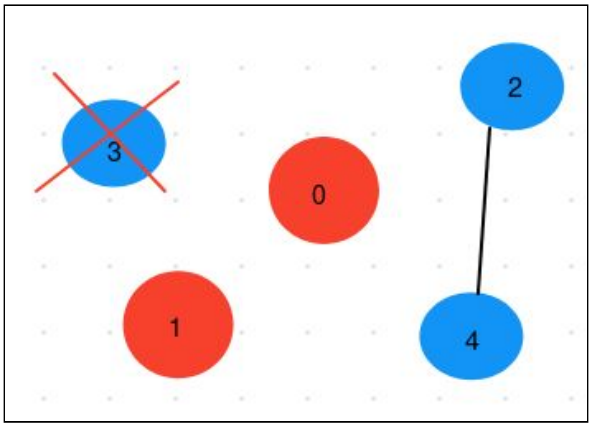
Se añade a solución el nodo 0 y el grado del nodo 0 en el vector de candidatos se pone a 0, reduciendo el grado de sus nodos adyacentes (el nodo 1 y nodo 2).

Vector solución	{0}
Vector candidatos	{0, 1, 1, 1, 1}



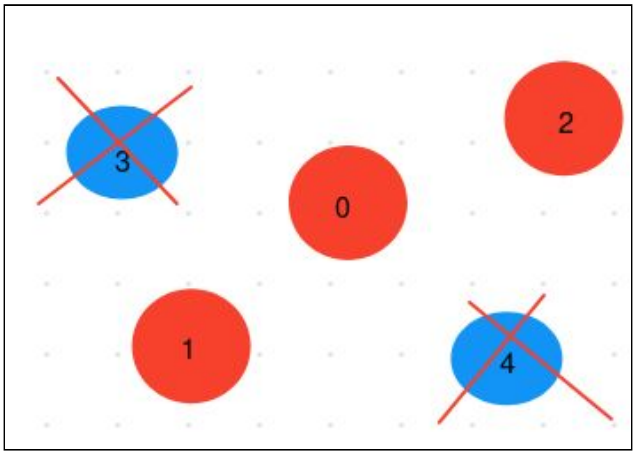
Se repite la búsqueda en el vector de candidatos para el nodo con mayor grado, siendo elegido el nodo 1, estableciendo su grado a 0 y reduciendo el grado del nodo adyacente en 1.

Vector solución	{0, 1}
Vector candidatos	{0, 0, 1, 0, 1}



En este punto se puede ver que se han añadido dos nodos a la solución {0, 1} ,y que un nodo candidato (3) se ha quedado con grado 0. Esto quiere decir que dicho nodo no es candidato a solución ya que su recorrido ya ha sido recubierto, por lo que para las siguientes búsquedas, dicho nodo será excluido por tener grado 0.
El siguiente nodo elegido es el nodo 2, que se añade a solución, quedando el vector de candidatos con todos los nodos con grado 0 (no hay más candidatos).

Vector solución	{0, 1, 2}
Vector candidatos	{0, 0, 0, 0, 0}



Por tanto, se obtiene un vector resultado con los nodos {0, 1, 2}. Como se puede ver claramente, este recubrimiento no es minimal ya que escogiendo los nodos 1 y 2 el grafo se recubrimiento minimalmente.

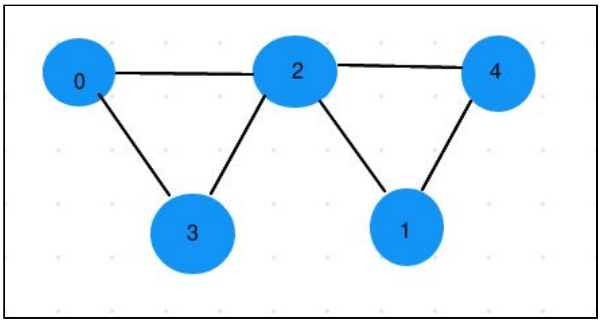
ii. Pruebas de ejecución

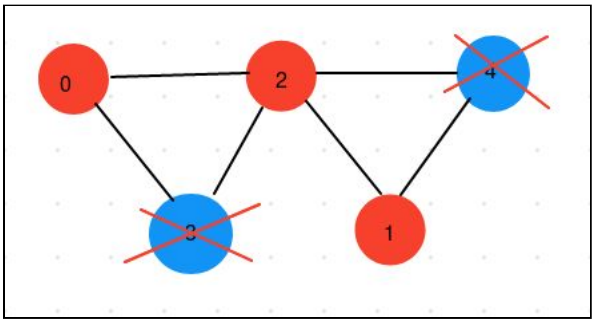
A continuación vamos a mostrar diferentes grafos y la solución obtenida (nodos rojos), todos ellos disponibles en la carpeta “grafo” ubicada en “data” (data/grafo/).

En las imágenes siguientes se pueden observar los resultados para los distintos archivos de grafos:

Grafo 1

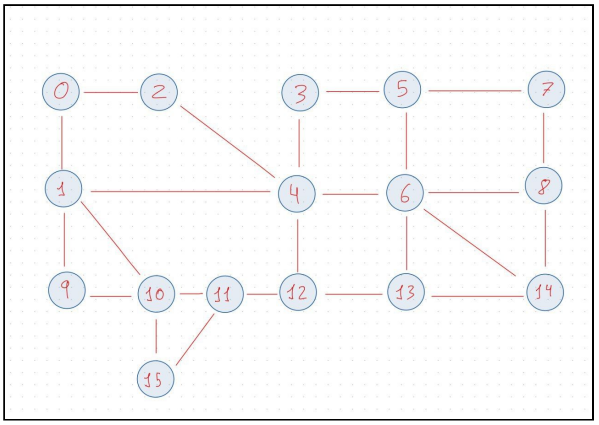
```
nonsatus@Non [29/04/20 21:26:11 miércole
$./bin/grafo data/grafo/grafo1.gra
Solucion: 2 0 1
Tiempo: 4.785e-06
```

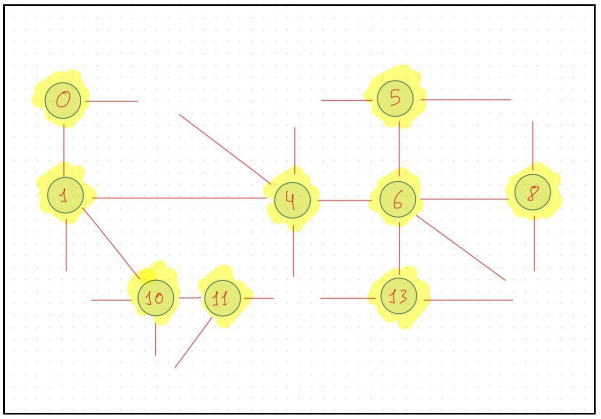




Grafo 2

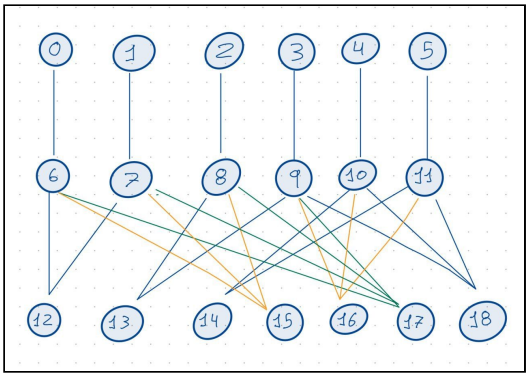
```
nonsatus@Non [29/04/20 21:26:13 miércoles]:
$./bin/grafo data/grafo/grafo2.gra
Solucion: 4 6 10 0 5 8 11 13 1
Tiempo: 2.1305e-05
```

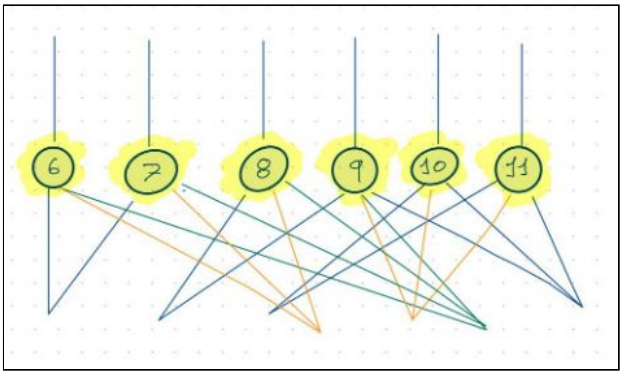




Grafo 3

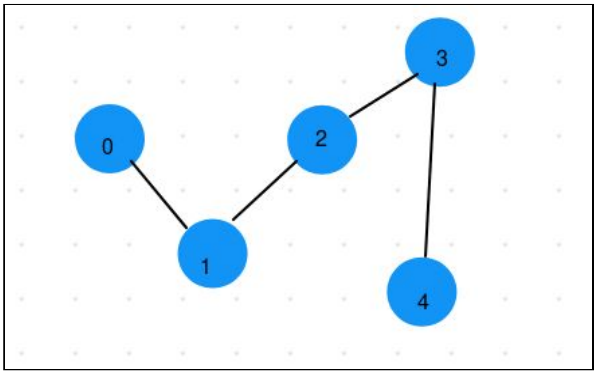
```
nonsatus@Non [29/04/20 21:26:16 miércoles]
$./bin/grafo data/grafo/grafo3.gra
Solucion: 7 8 9 6 10 11
Tiempo: 1.6479e-05
```

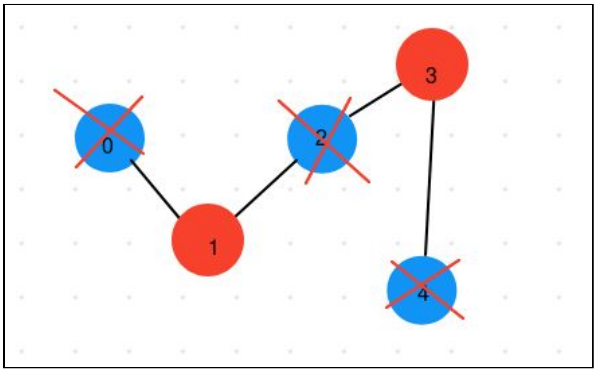




Grafo 4

```
nonsatus@Non [29/04/20 21:26:59 miércoles]
$./bin/grafo data/grafo/grafo4.gra
Solucion: 1 3
Tiempo: 5.338e-06
```

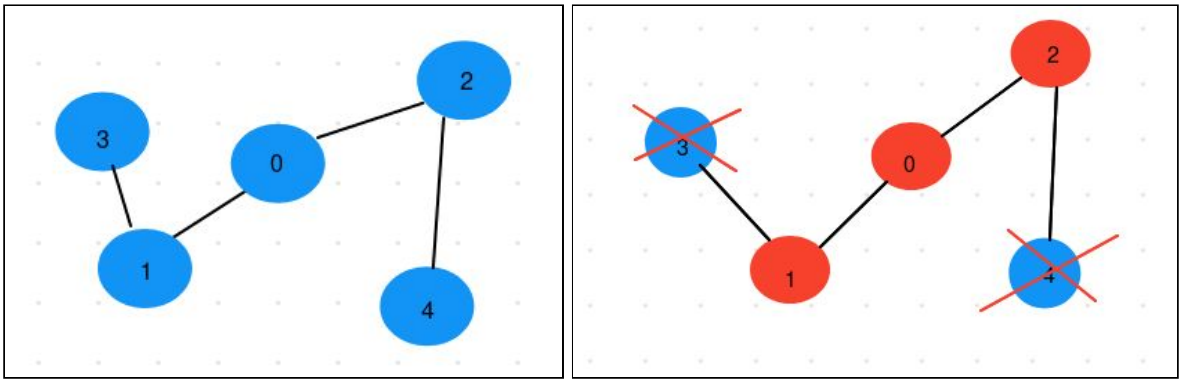




Grafo 5

En este caso se ha demostrado anteriormente que la solución no es minimal.

```
nonsatus@Non [29/04/20 21:27:04 miércoles
$./bin/grafo data/grafo/grafo5.gra
Solucion: 0 1 2
Tiempo: 6.831e-06
```



Como se puede ver, el grafo 4 y el grafo 5 tienen la misma estructura, pero distinta organización de nodos, así se puede comprobar que según la organización de los nodos, el resultado es diferente, pudiendo o no dar el recubrimiento minimal, el cual se obtiene en el grafo 4 pero no en el grafo 5.

b. Árbol

En este apartado se va a abordar el mismo problema anterior solo que en vez de ser para grafos, será para el caso específico de árboles, la diferencia entre ambos es que en un árbol no existen ciclos, esto es, que recorriendo el árbol no se puede llegar al mismo punto origen. También destacar que en un árbol, para llegar a un nodo determinado, solo se puede llegar unicamente a traves de un nodo concreto, y no a través de otro nodo.

Así, mientras que en un grafo se indicaba para cada nodo su conexión con los demás nodos, en este caso se especifica un nodo "padre" y que nodos "hijos" dispone dicho padre.

Características Greedy

Conjunto candidatos	Todos los nodos
Conjunto seleccionados	Nodos padres de nodos con grado 1
Función solución	Nodos que recubren minimalmente el árbol
Función selección	Nodo con grado 1
Función factibilidad	No hay más nodos con grado 1
Función objetivo	Padre del nodo de grado 1

Pseudocódigo

1 - Leer árbol de fichero
2 - Generar matriz de adyacencia
3 - Generar vector de candidatos (cada posición refleja un nodo y el valor el grado de dicho nodo)
4 - Mientras queden nodos con grado mayor a 0
5 - Si el nodo tiene grado 1
6 - Busca el nodo padre del nodo candidato
7 - Si encuentra el padre
8 - Comprueba que no esté ya en la solución
9 - Si no está en la solución
10 - Se actualiza el nodo padre candidato
11 - Añade el padre a la solución
12 - Reduce el grado de los nodos adyacentes (padre del padre e hijos del padre) en 1
13 - Reduce el grado del nodo solución a 0
14 - Reinicializa la posición de búsqueda en el vector de candidatos
15 - Aumenta la posición del nodo a comprobar

(Haciendo referencia al pseudocódigo)

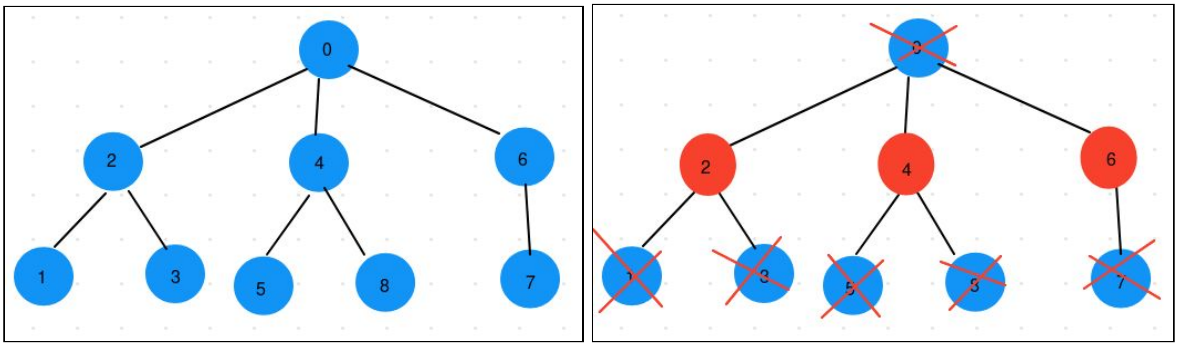
En el peor de los casos este algoritmo tiene una eficiencia $O(n^3)$ ya que, tendría que recorrer todos los nodos “n” veces (O(n)) (“while” linea 4), para cada nodo comprobado busca el padre en la matriz de adyacencia (“for” linea 6) (recorre la fila de la matriz), que en el peor caso tiene que recorrerla entera (O(n)) y para cada posible padre del nodo tiene que comprobar en el vector solución si dicho nodo está ya insertado (O(n)). Como se tiene tres bucles anidados cuyo orden de eficiencia de cada bucle es O(n), la eficiencia en el peor de los casos es $O(n^3)$.

i. Pruebas de ejecución

A continuación se van a mostrar distintas ejecuciones con sus soluciones minimales obtenidas.

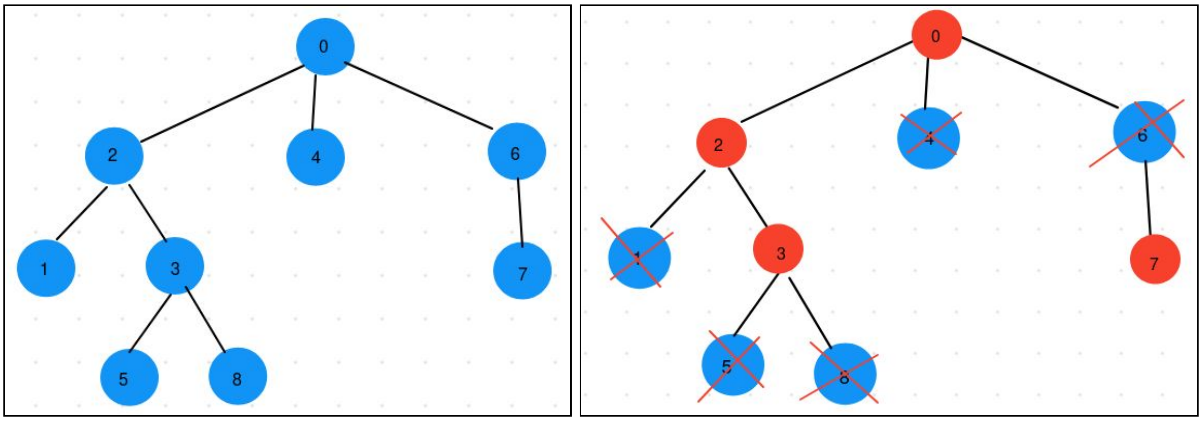
Árbol 1

```
nonsatus@Non [30/04/20 11:37:04 jueves]:
$./bin/arbol data/arbol/arbol1.arb
Solucion: 2 4 6
Tiempo: 4.332e-06
```



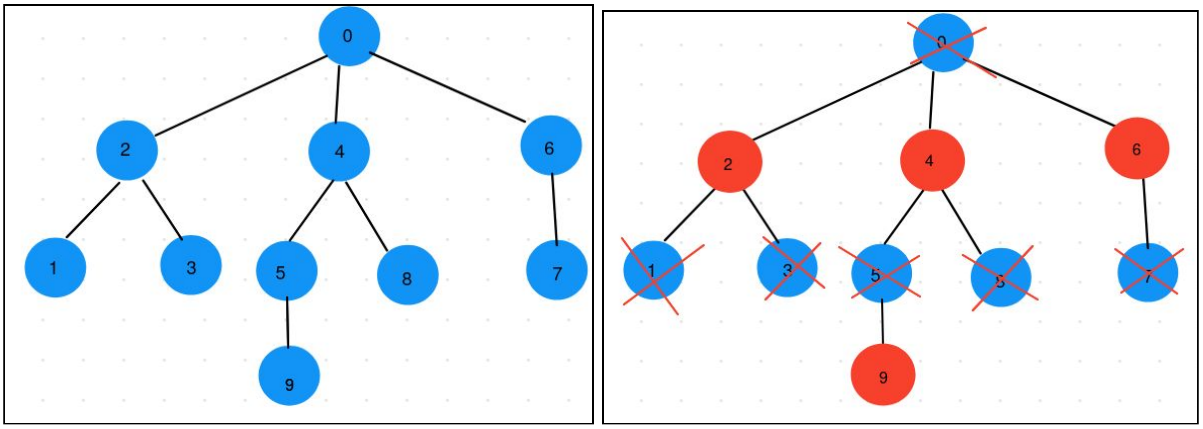
Arbol 2

```
nonsatus@Non [30/04/20 11:37:06 jueves]
$./bin/arbol data/arbol/arbol2.arb
Solucion: 2 0 3 7
Tiempo: 3.307e-06
```



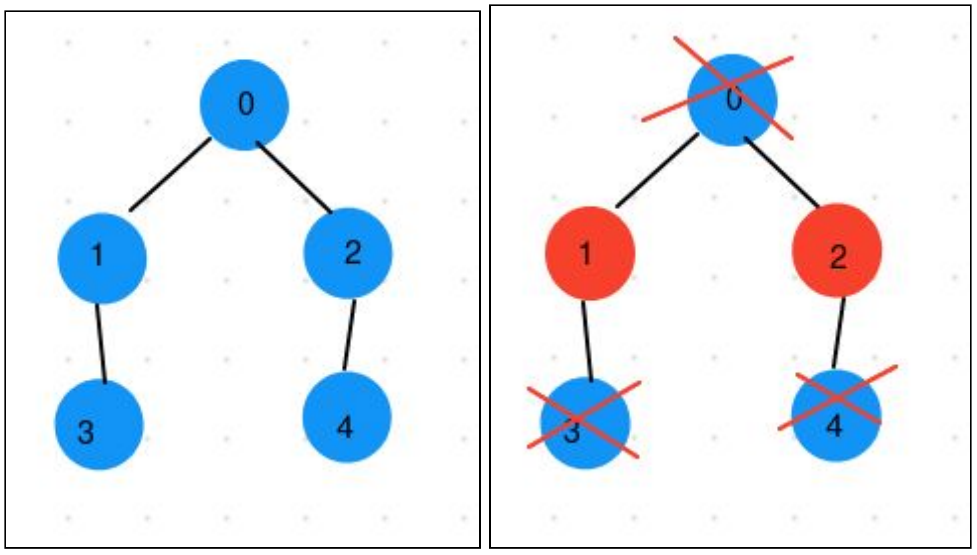
Árbol 3

```
nonsatus@Non [30/04/20 11:37:08 jueves]
$./bin/arbol data/arbol/arbol3.arb
Solucion: 2 6 4 9
Tiempo: 6.112e-06
```



Árbol 4

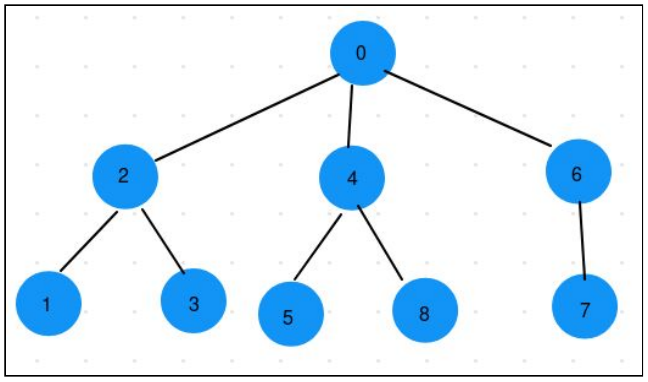
```
nonsatus@Non [30/04/20 11:37:11 jueves]
$./bin/arbol data/arbol/arbol4.arb
Solucion: 1 2
Tiempo: 3.275e-06
```



Como se puede ver, para todos los ficheros de pruebas para árboles se han obtenido el recubrimiento minimal de cada árbol, nótese que el árbol 4 es la misma representación que el grafo 5, en el cual, mediante una heurística para grafos, no se había obtenido el recubrimiento minimal.

ii. Ejemplo de ejecución

Se va a mostrar el funcionamiento del algoritmo paso a paso con el ejemplo del árbol 1.



Vector solución	{vacío}
Vector candidatos	{3, 1, 3, 1. 3, 1. 2. 1, 1}

Cada posición del vector candidatos hace referencia a un nodo (Posición 0 referencia al nodo 0, indicando que tiene un grado 3)

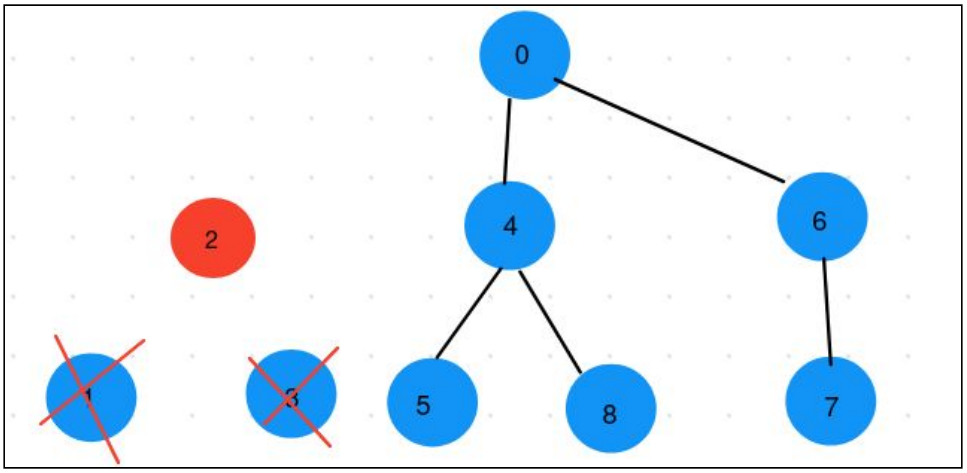
Primero se busca el primer nodo con grado 1, en este caso, corresponde con el nodo 1.

Se busca en la matriz de adyacencia el padre (se busca en la fila del nodo 1 y cuando se encuentre una conexión indicada con un 1, se obtiene el padre, representado en la columna), obteniendo que el padre es el nodo 2.

En azul se representan los nodos (Padres e hijos), en rojo claro se representa el vector de candidatos, donde la posición del vector representa el nodo y el valor el grado del nodo, las casillas verdes indican una conexión entre un nodo hijo y un nodo padre.

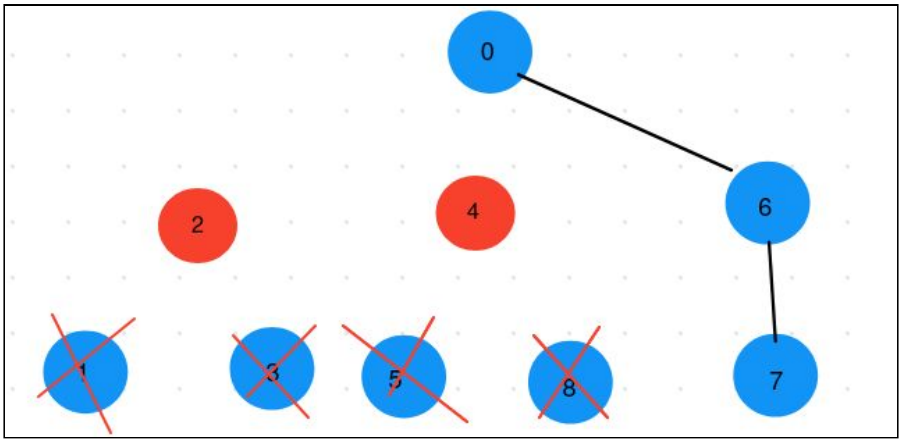
Nodo	0	1	2	3	4	5	6	7	8	Grado
0	0	0	1	0	1	0	1	0	0	3
1	0	0	1	0	0	0	0	0	0	1
2	1	1	0	1	0	0	0	0	0	3
3	0	0	1	0	0	0	0	0	0	1
4	1	0	0	0	0	1	0	0	1	3
5	0	0	0	0	1	0	0	0	0	1
6	1	0	0	0	0	0	0	1	0	2
7	0	0	0	0	0	0	1	0	0	1
8	0	0	0	0	1	0	0	0	0	1

Se añade el nodo 2 al vector solución, se reduce el grado del nodo 2 a 0 y en 1 el padre del nodo 2 y los hijos del nodo 2, quedando:



Vector solución	{2}
Vector candidatos	{2, 0, 0, 0. 3, 1. 2. 1, 1}

Se vuelve a recorrer el vector de candidatos buscando el primer nodo de grado 1, obteniendo el nodo 5. Se busca en la matriz de adyacencia el padre y se obtiene el nodo 4, el cual se añade a la solución y se reduce su grado a 0 y el grado de sus adyacentes en 1.



Vector solución	{2, 4}
Vector candidatos	{1, 0, 0, 0. 0, 0. 2. 1, 0}

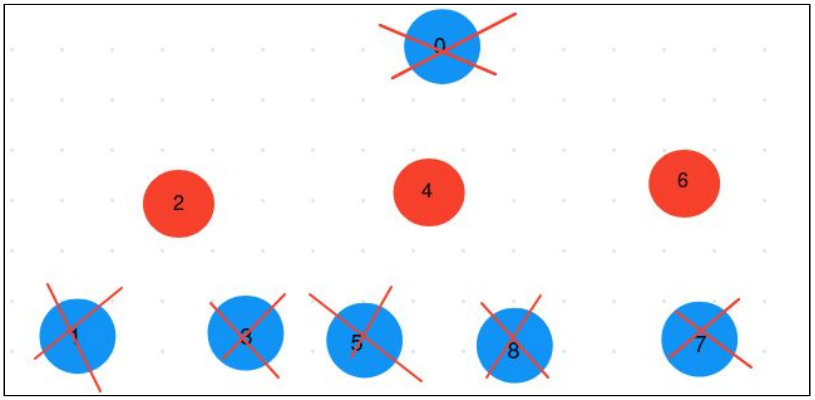
Ahora vuelve a buscar un nodo de grado 1, escoge el nodo 0, busca en la matriz de adyacencia el padre, el primer nodo encontrado con conexion es el nodo 2, pero como este nodo ya forma parte de la solución, se descarta el nodo 2 y se busca el siguiente nodo como posible padre, que sería el 4, que también es solución y repite hasta encontrar el nodo 6, el cual no es solución.

Se añade a solución y después se establece el grado del nodo 6 a 0 y se reduce en 1 el de sus adyacentes (el nodo 0 y el nodo 7), quedando el vector de candidatos con todos los nodos con grado 0.

Aquí hay que contemplar que hay casos en los que al quedar una rama de un árbol, esta se puede contemplar de diferente forma

- El nodo 0 es el padre del nodo 6 y el nodo 6 el padre del nodo 7
- El nodo 6 es el padre del nodo 0 y nodo 7
- El nodo 7 es el padre del nodo 6 y el nodo 6 el padre del nodo 0

Así, de cualquier forma en la que se interprete, la solución es la misma, es decir, realmente no se habla de padres e hijos sino de nodos adyacentes, pero se usa el término “padre” e “hijo” ya que por la lógica de la estructura es más fácil reconocerlos de esa forma.



Vector solución	{2, 4, 6}
Vector candidatos	{0, 0, 0, 0. 0, 0. 0. 0, 0}

c. Ficheros de pruebas

Para representar la estructura de un grafo se han usado las siguientes directrices:

- Respecto al fichero de datos de grafos
    - Fichero debe comenzar con “SOF” (Start Of File)
    - Cada línea representa un nodo seguido de los nodos adyacentes
        - Ejemplo, la línea “2 1 3” indica que el nodo 2 es adyacente al nodo 1 y 3.
    - Fichero debe terminar con “EOF” (End Of File)
- El identificador de los nodos es entero y tiene que ir del 0 hasta N_nodos-1

Para representar la estructura de un árbol se han usado las siguiente directrices:

- Respecto al fichero de datos de árbol
    - Fichero debe comenzar con “SOF” (Start Of File)
    - Cada línea representa un nodo padre seguido de los nodos hijos
        - Si no tiene hijos, el nodo solo se refleja como hijo de otro nodo
        - Ejemplo, la línea “2 1 3” indica que el nodo 2 es padre del nodo 1 y 3.
    - Fichero debe terminar con “EOF” (End Of File)
- El identificador de los nodos es entero y tiene que ir del 0 hasta N_nodos-1

d. Comparativa

Nuestro algoritmo de grafos funciona tanto para grafos como para árboles (ya que un árbol es un grafo pero un grafo no es árbol), pero nunca nos asegura que el recubrimiento que devuelve puede ser minimal.

Por otro lado nuestro algoritmo de árboles, no funciona para grafos que no sean árboles, pues puede darse el caso de que no se haya terminado de quitar nodos pero los nodos que queden no sea ninguno de grado uno (esto no pasa en árboles nunca); sin embargo, sí que nos asegura que el recubrimiento encontrado es minimal.

Por ejemplo, al aplicar ambos algoritmos al contraejemplo dado anteriormente, vemos que el algoritmo de grafos falla mientras que el algoritmo de árboles nos devuelve el recubrimiento minimal.

5. Conclusiones

Debido a las características greedy, una solución óptima no siempre está asegurada, ya que en cada paso del problema se elige la mejor opción que en ese momento esté disponible, sin pensar si elegir una opción más mala pudiera resultar en una solución global mejor.

Por tanto, la solución obtenida va a depender de dos factores:

- La heurística elegida para la resolución
- El tipo del problema a resolver

Así se puede ver que en el problema del viajante de comercio las soluciones obtenidas no siempre son las óptimas (puede dar la casualidad de que la de) pero sí se consigue una aproximación, que variará en función de la heurística (vecino más cercano, inserción y 2-opt) siendo la heurística del vecino más cercano la que peor resultado da, y la heurística 2-opt una aproximación bastante cercana (pero no óptima).

En el caso del recubrimiento minimal para grafos no dirigidos, se ha podido ver que para el caso de que sea un grafo el recubrimiento no está asegurado mientras que en el caso particular de un árbol, el recubrimiento minimal si se ha obtenido.

6. Bibliografía

Lecciones de Algorítmica - Verdegay Galdeano, José Luis
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://es.wikipedia.org/wiki/Algoritmo_del_vecino_m%C3%A1s_pr%C3%B3ximo
https://en.wikipedia.org/wiki/2-opt