UNIVERSIDAD DE GRANADA

Práctica 3: Greedy

Algoritmos voraces

José María Gómez García
Fernando Lojano Mayaguari
Valentino Lugli
Carlos Mulero Haro

Universidad de Granada
Granada
Abril 2020

Índice

1 Introducción

En esta práctica realizaremos un estudio y análisis sobre los algoritmos voraces, denominados algoritmos Greedy. Para llevar a cabo este estudio, presentaremos cuatro algoritmos, tres de ellos resuelven el mismo problema, conocido como Travelling Salesman Problem (TSP) o el problema del viajante de comercio. El último algoritmo soluciona el clásico problema de Planificación de tareas. No obstante, para poder meternos un poco en el contexto y comprender mejor el enfoque de los algoritmos voraces, daremos primero una explicación primitiva sobre el mismo.

Los algoritmos Greedy o voraces se caracterizan por escoger en cada momento el mejor caso presente y que mayor beneficio otorgue sin tener en cuenta ni preocuparse por las posibles repercusiones que se puedan dar al tomar dichas decisiones en el momento. Como consecuencia de esto, Greedy nos otorga LA MAYORIA DE LAS VECES el mejor resultado que se puede obtener. Esto significa que en la mayoría de los casos, la solución que se obtiene con el enfoque voraz es la solución óptima del problema. Normalmente este tipo de enfonque se utiliza para resolver problemas de optimización. Existen seis características que debe reunir un problema para poder ser resuelto por este enfoque:

- **Lista de candidatos:** Conjunto de candidatos iniciales en el problema
- **Lista de candidatos utilizados:** Conjunto de candidatos ya usados
- **Función solución:** Una función solución que indica cuando un conjunto de candidatos es una solución al problema, no necesariamente óptima.
- **Función selección:** Una función de selección que nos indica cual es el mejor candidato no usado en cada momento.
- **Función de factibilidad:** Un criterio para analizar si un conjunto de candidatos es factible
- **Función objetivo:** Una función objetivo, que es aquella que intentamos optimizar.

El modo de funcionamiento general de el enfoque greedy consiste en: Escoger del conjunto de candidatos disponibles el mejor de ellos según la función de selección y analizar si este cumple las condiciones de factibilidad dentro del conjunto solución, que inicialmente se encuentra vacío. Si el candidato no cumple dichas condiciones, se elimina de la lista de candidatos disponibles y no se volverá a analizar más, en caso contrario, se incluye el candidato en el conjunto solución y lo se elimina de los candidatos disponibles. Una vez hecho esto, queda por analizar si estamos ante una solución evaluando la función objetivo. Si no se diese el caso, se repite el proceso hasta llegar a una solución óptima.

Ahora que hemos dado una breve explicación de este enfoque, pasaremos a analizar cada uno de los algoritmos propuesto de una forma más extensa, empezando por aquellos tres que se encargan de encontrar una solución al problema del viajante de comercio (problema TSP).

2 Problema TSP

2.1 Basado en cercanía

2.1.1 Descripción

Le heurística del algoritmo Greedy para la resolución del problema del vendedor viajante por medio de cercanía es una bastante directa, dada una ciudad v_0 se toma una ciudad v_i que no se encuentre en el circuito y que posea la menor distancia. Esto se repite hasta que no queden más ciudades.

Esto se traduce a que, se posee una lista de ciudades C con sus coordenadas X e Y, de todas ellas se elige una ciudad de origen que se elimina C y pasa a formar parte de la lista solución T. Seguidamente, se elige del resto de ciudades en C aquella que posea la menor distancia hacia la ciudad de origen, la cual se añade a la solución T, se elimina de los candidatos C, se suma la distancia del tramo y se repite el ciclo ahora buscando otra ciudad que sea más cercana a ésta última elegida hasta que no queden más ciudades en C.

Al finalizar, se calcula la distancia entre la última ciudad en la solución y la ciudad de origen para finalmente generar el camino hamiltoniano.

2.1.2 Componentes del Algoritmo Voraz

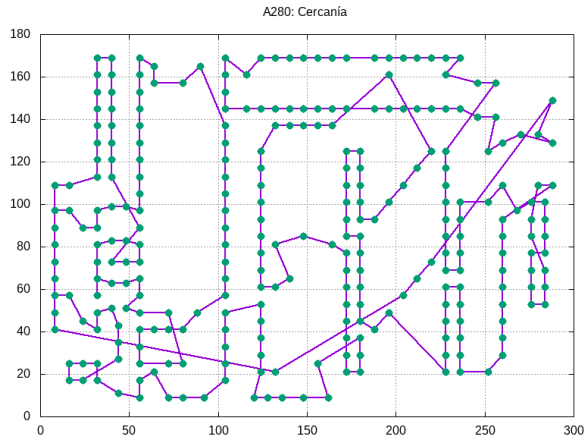
- **Lista de candidatos:** Nodos del grafo original
- **Lista de candidatos utilizados:** Aquellos nodos que ya han sido seleccionados para formar parte de la solución.
- **Función solución:** El número de nodos en la solución es el número de nodos del grafo.
- **Función selección:** Se selecciona el nodo v_0 tal que la distancia con el último nodo ya visitado v_i sea el mínimo.
- **Función de factibilidad:** No se pueden formar ciclos, esto siempre se cumple ya que siempre se escoge un nodo nuevo.
- **Función objetivo:** Encontrar un ciclo hamiltoniano minimal del grafo.

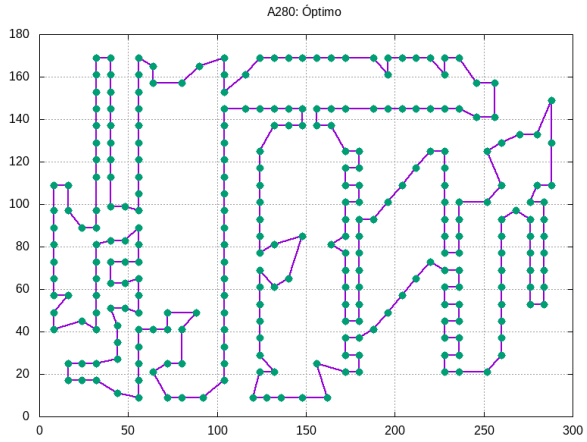
2.1.3 Pseudocódigo

```
funcion cercaniaTSP(G=(C,A), T=(V,A))
inicio
    nIni := Nodo cualquiera de C
    nFin := nIni
    nVec

    C.eliminar(nIni)    // C es ahora la lista de candidatos
    T.insertar(nIni)    // Solucion a crear

    mientras !C.esVacio() hacer
        nVec := seleccionar nodo en C donde distancia=(nVec,nIni) sea menor
        T.insertar(nVec)    // Insertar en solucion nodo mas cercano a nIni
        C.eliminar(nVec)    // Quitarlo de los candidatos
        nIni := nVec
    fmientras
    T.insertar(nIni)    // Cerrar el camino, insertando el nodo inicial y
       calculando distancia entre el ultimo nodo candidato y el inicial.
ffunc
```

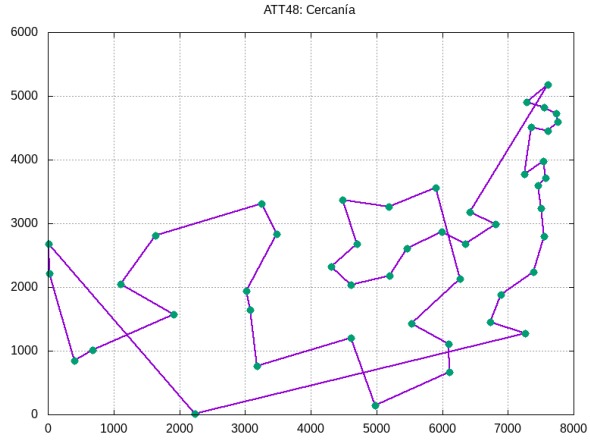
2.1.4 Escenarios de ejecución

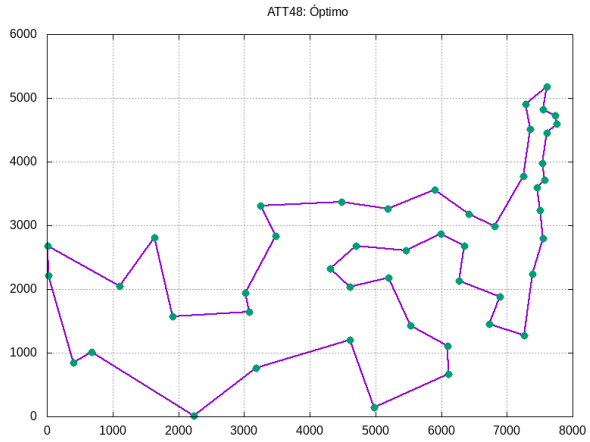
El código se probó con todos los escenarios previstos sin ningún problema, dando una solución para cada uno, naturalmente sub-óptima. Aquí se compara la ruta hecha por el algoritmo Cercanía contra la ruta óptima. Algo importante a destacar de los gráficos es que la escala de los ejes es diferente, permiten visualizar de una manera más cómoda el recorrido pero una consecuencia de esto es que pueden haber puntos que si bien se ven más cercanos entre sí, realmente no lo están.



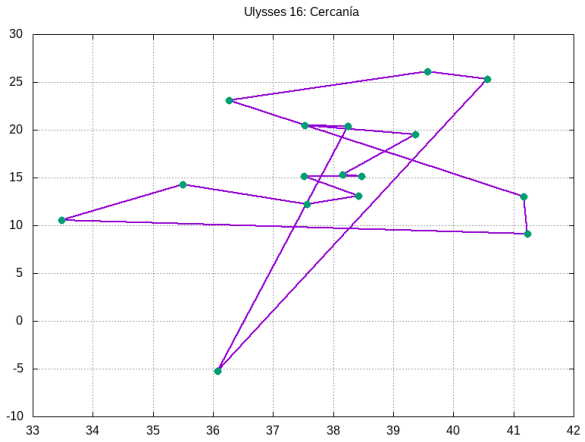


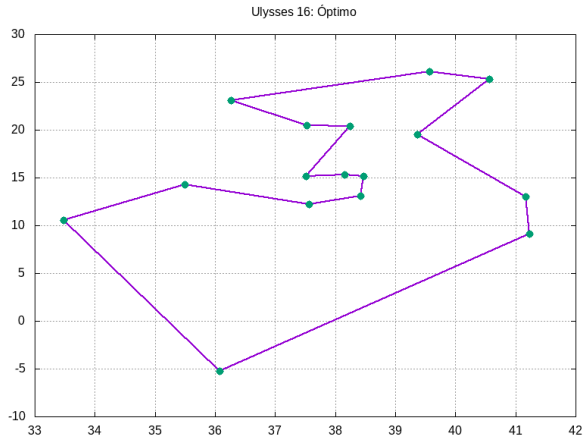
A280



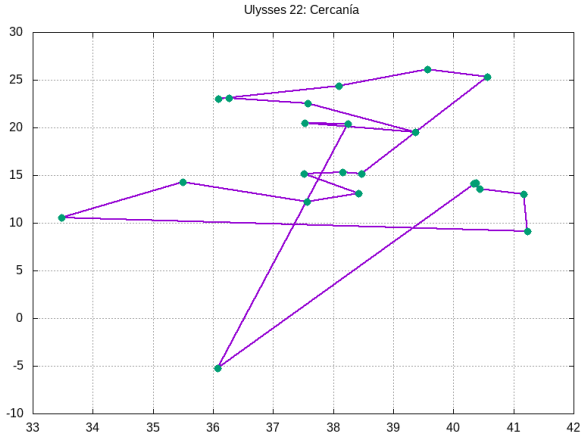


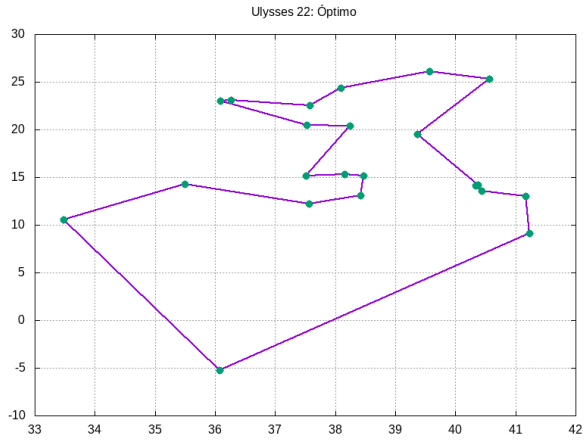
ATT48





Ulysses 16





Ulysses 22

TSP	Óptimo	Cercanía	
		Camino	Diferencia
A280	2579	3157	22,41%
ATT48	33522	40583	21,06%
Ulysses 16	73	103	41,1%
Ulysses 22	74	93	25,68%

Si bien hay ciertas similitudes con los recorridos óptimos, la distancia es bastante mayor, naturalmente el problema que posee este enfoque es que si bien al principio el algoritmo puede obtener las distancias más pequeñas entre dos ciudades, a medida que avanza, al quedar menos ciudades disponibles empieza a realizar recorridos más y más largos, lo cual empieza a sumar más distancia de la necesaria, cuando se encuentra en la última ciudad y va a devolverse al inicio, este último paso puede ser, dependiendo de dónde a terminado el algoritmo una distancia considerable.

Aún así, realizando una comparativa entre los datos óptimos del camino y los obtenidos por cercanía, la diferencia entre el ambos oscila en un $27,6\%$. Es decir, el camino por cercanía es en promedio un $27,6\%$ más largo que el óptimo, por lo que, teniendo en cuenta que el algoritmo Greedy realiza esto en tiempo polinomial $O(n^2)$ mientras que la solución óptima se obtiene en tiempos no polinomiales dada la naturaleza NP-Completa, se puede decir que es una aproximación relativamente buena.

2.2 Basado en inserción

2.2.1 Descripción

En la solución del problema del viajante de comercio basada en inserción, realizamos inicialmente un recorrido parcial que abarca tres puntos del recorrido total a tratar, concretamente los nodos situados mas al norte, al este y al oeste del mapa según sus coordenadas. Una vez formado este recorrido, nuestro algoritmo voraz va insertando al recorrido el nodo mas cercano a cualquiera de los ya incluidos y se colocara justo antes de este. Realizando esta operación sucesivamente hasta recorrer todos los nodos obtendremos finalmente un grafo con la solución final.

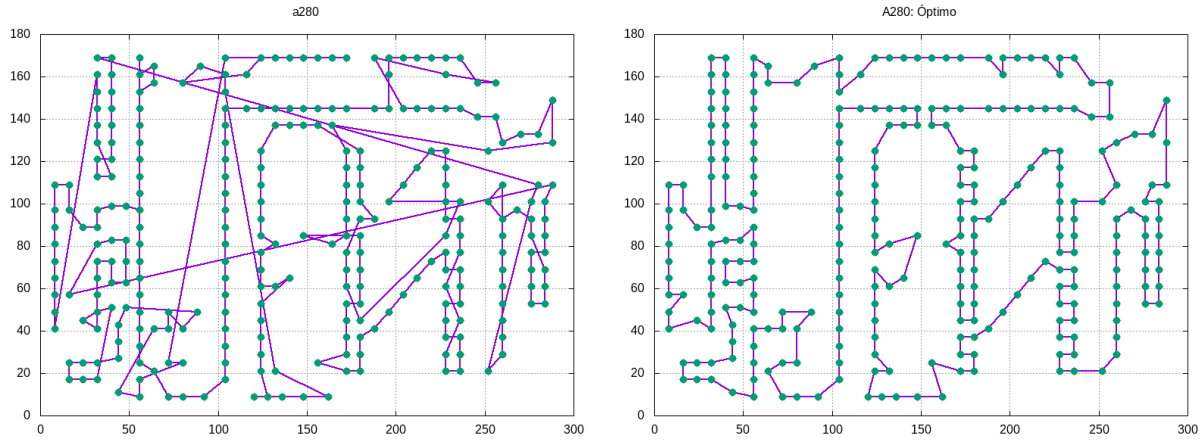
2.2.2 Componentes del Algoritmo Voraz

- **Lista de candidatos:** Nodos del grafo original, menos los posicionados mas al norte, mas al este y mas al oeste según las coordenadas marcadas.
- **Lista de candidatos utilizados:** Aquellos nodos que según el criterio de selección pasan a formar parte de la solución.
- **Función solución:** El número de nodos en la solución es el número de nodos del grafo.
- **Función selección:** Devuelve el nodo a insertar en la siguiente iteración y el nodo delante del cual debe insertarse.
- **Función de factibilidad:** No se pueden formar ciclos, ya que el nodo seleccionado para ser insertado en la solución se elimina de los candidatos.
- **Función objetivo:** Encontrar un ciclo hamiltoniano de peso mínimo para el grafo.

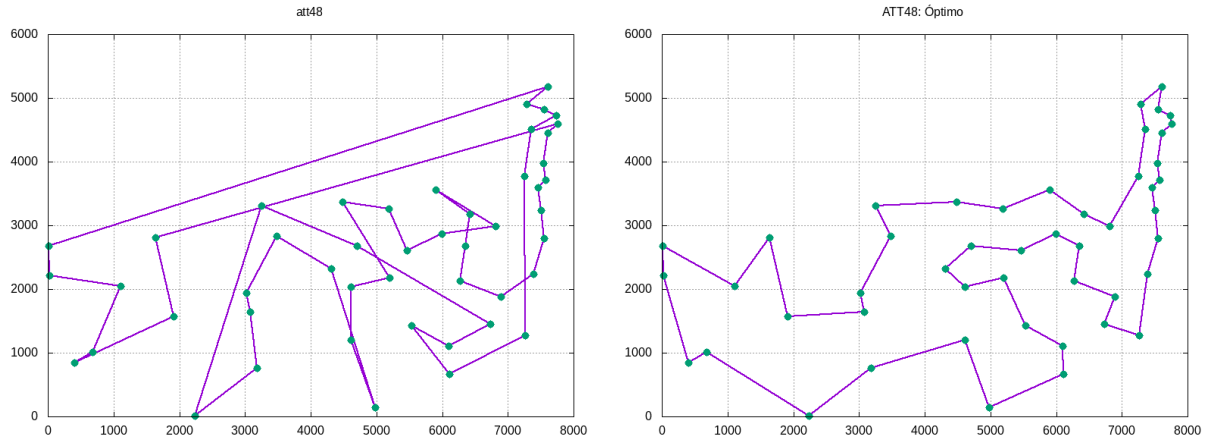
2.2.3 Pseudocódigo

```
funcion insercionTSP(G, dist)
inicio
    dist := G.nodoNorte().distancia(G.nodoEste())+G.nodoEste().distancia(G.
       nodoOeste())
    g_res //Grafo donde se recoge la solucion

    g_res = G.recorridoParcial() //formamos el recorrido parcial con los tres
       nodos iniciales
    //Eliminamos de los candidatos los nodos del recorrido inicial
    G.eliminar(G.nodoEste())
    G.eliminar(G.nodoOeste())
    G.eliminar(G.nodoNorte())
    aux //pareja donde almacenamos el nodo seleccionado y el nodo delante del
       cual insertamos el seleccionado

    mientras !G.vacio() hacer
        aux := seleccionar nodo en C donde distancia sea menor
        g_res.insertar(aux.first, aux.second)    // Insertar en solucion aux.
           first delante de aux.second
        C.eliminar(aux.second)    // Quitarlo de los candidatos
        dist+=aux.first.distancia(aux.second)
    fmientras

    T.insertar(g_res.primero)    // Cerrar el camino, insertando el nodo
       inicial y calculando distancia entre el ultimo nodo candidato y el
       inicial.
    devolver g_res
ffunc
```

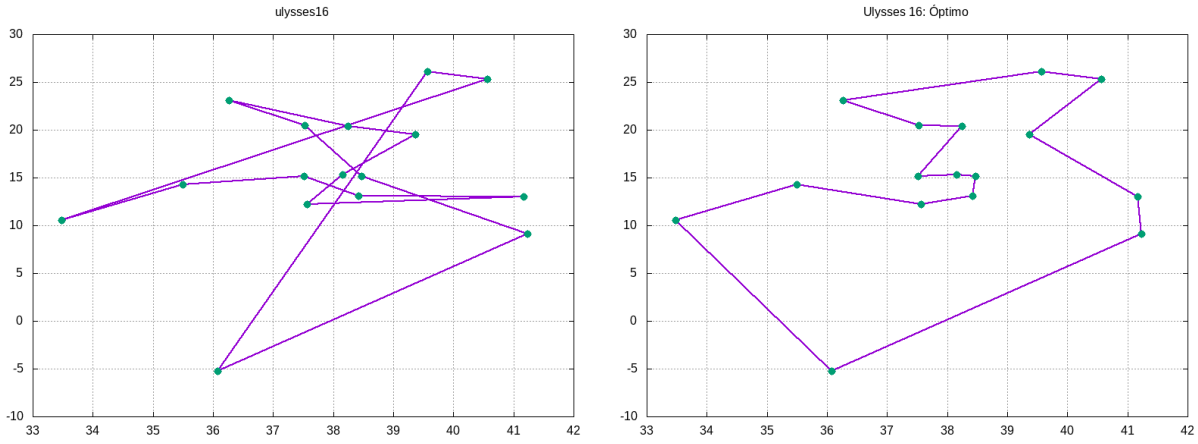
2.2.4 Escenarios de ejecución



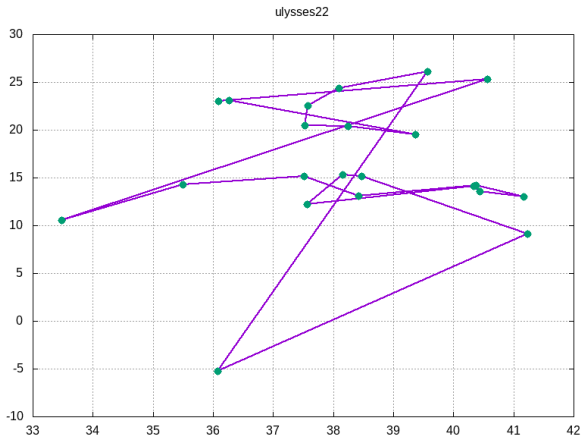
A280

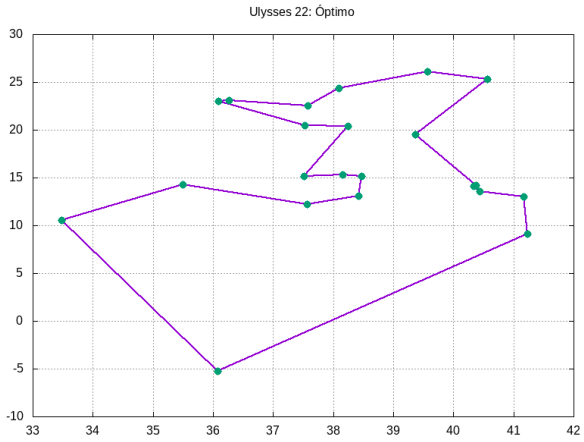


ATT48



Ulysses 16





Ulysses 22

TSP	Óptimo	Inserción	
		Camino	Diferencia
A280	2579	4179	62,04%
ATT48	33522	55982	67,00%
Ulysses 16	73	105	43,83%
Ulysses 22	74	107	44,59%

Podemos observar que los resultados obtenidos por este algortimo son algo peores que en el caso de la resolución por cercanía. Esto se puede ver reflejado tanto en las distancias obtenidas como en el porcentaje de diferencia con respecto a los resultados óptimos, teniendo casos en los que se supera el 60% de diferencia. Podemos afirmar por tanto que, de los dos métodos de resolución que se nos proporciona para este análisis, el que mejores resultados nos ofrece es el método de resolución por cercanía.

2.3 Nuestra solución

2.3.1 Descripción

En nuestra solución hemos resuelto el problema TSP teniendo en cuenta la distancia entre nodos, esta distancia en el gráfico forma una arista entre dos nodos por lo que hemos hemos decido llamar a esta resolución, "resolución por aristas".

Para llevar a cabo esta forma de resolverlo hemos de calcular primero la distancia que hay entre todos los nodos; a continuación ordenamos todas estas distancias (cada distancia es la arista entre un nodo 'a' y un nodo 'b'); seleccionamos la primera distancia de la lista, que será la menor, e iremos insertando nodos a la solución teniendo en cuenta que es el nodo a insertar crea la arista de menor tamaño a añadir a nuestra solución.

2.3.2 Componentes del Algoritmo Voraz

- **Lista de candidatos:** Todas las parejas de nodos del grafo original posibles (distancia entre los nodos, la cual llamaremos "arista").
- **Lista de candidatos utilizados:** Todos los nodos que forman parte de las parejas seleccionadas como solución.
- **Función solución:** El número de nodos de nuestra solución es el mismo que el número de nodos original; siendo el último igual que el primero para cerrar el camino.
- **Función selección:** Se selecciona la pareja de nodos v_0 tal que la distancia entre ellos sea la mínima y cualquiera de sus nodos se pueda unir a la solución.
- **Función de factibilidad:** No se forman ciclos ya que las parejas disponibles que podrían crearlos son eliminadas al insertar un nuevo nodo a la solución.
- **Función objetivo:** Encontrar un ciclo hamiltoniano de peso mínimo para el grafo.

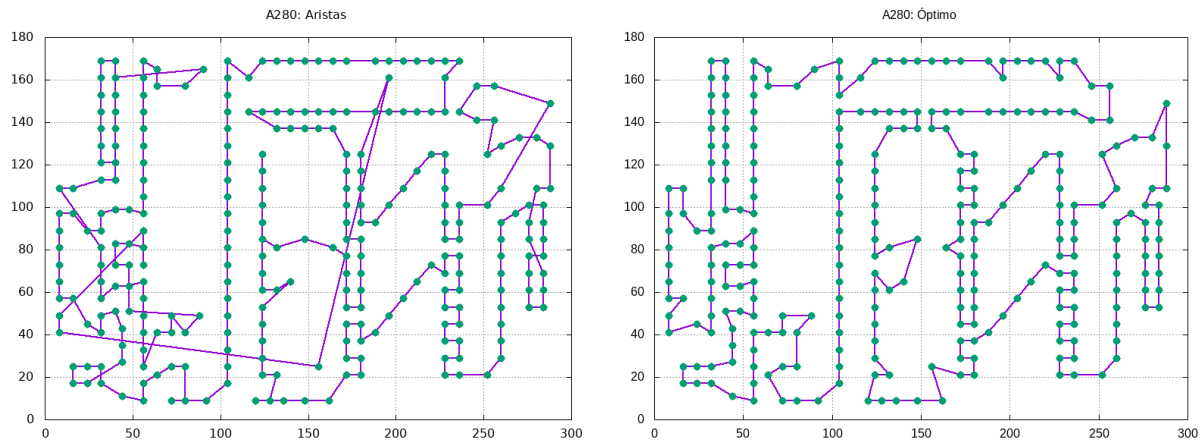
2.3.3 Pseudocódigo

```
funcion aristasTSP(x[], y[])
inicio
    dimension := x.size() // Numero de nodos candidatos

    // Calculamos las distancias entre todas las aristas
    para i:=0 hasta dimension-1 hacer
        para j:=i hasta dimension-1 hacer
            si i != j entonces
                distancias.push(calculardistancia(x[i],x[j],y[i],x[j])
                aristas_aux.push(i)
                aristas_aux.push(j)
            fsi
        fpara
    fpara

    // Ordenamos las distancias de menor a mayor distancia
    mientras aristas_sol.size() sea menor que (dimension-1)*dimension hacer

        indicemenordistancia := posicion del menor elemento de distancias[]

        aristas_ord.push(aristas_aux[indicemenordistancia*2])
        aristas_ord.push(aristas_aux[indicemenordistancia*2+1])
        distancias.eliminar(indicemenordistancia)
        aristas_aux.eliminar(indicemenordistancia*2,indicemenordistancia*2+1)
    fmientras
```

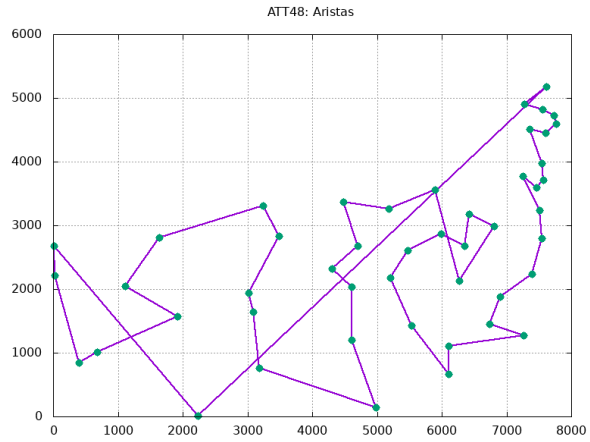
```
  // Incluimos la primera arista a nuestra solucion (la pareja de nodos cuya
     distancia es la menor se encuentra la primera ya que ordenamos
     previamente las distancias)
   solucion.push(aristas_ord[0])
   solucion.push(aristas_ord[1])
   aristas_ord.eliminar(0,1)

   mientras solucion.size() sea menor que dimension hacer

       encontrado := false

       para i:=0 hasta dimension-1 y mientras que !encontrado  con paso 2
          hacer
           si cualquier componente de la pareja de nodos puede insertarse en
              solucion entonces
               encontrado := true
               nodo_union := nodo de la pareja de nodos que ya forma parte de
                  la solucion en alguno de sus extremos
               solucion.insert(el nuevo nodo que forma parte de la solucion)
           fsi
       fpara

       para i:=0 hasta aristas_ord.size() con paso 2 hacer
           si aristas_ord[i] es igual a nodo_union o aristas_ord[i+1] es igual
               a npdp_union entonces
               aristas_ord.eliminar(i,i+1)
           fsi
       fpara
   fmientras

   // Cerramos el camino uniendo el final con el principio
   solucion.push(solucion[0])

ffuncion
```

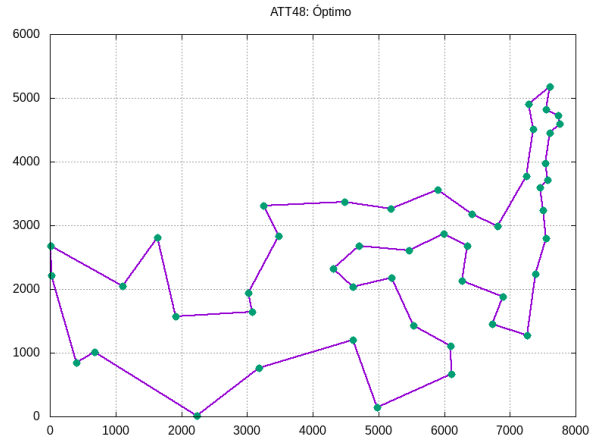
2.3.4 Escenarios de ejecución

Al probar la ejecución de nuestro algoritmo en los diferentes escenarios de ejecución estos fueron los resultados:

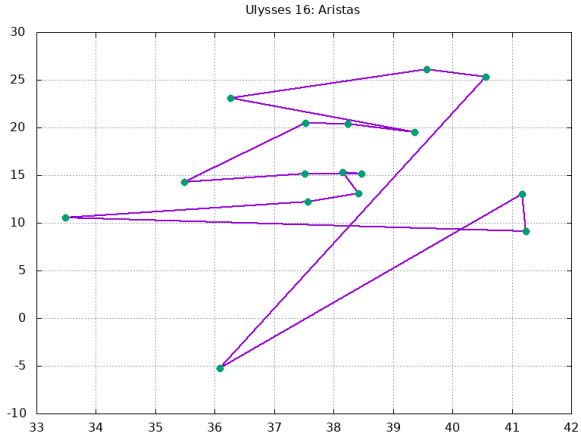


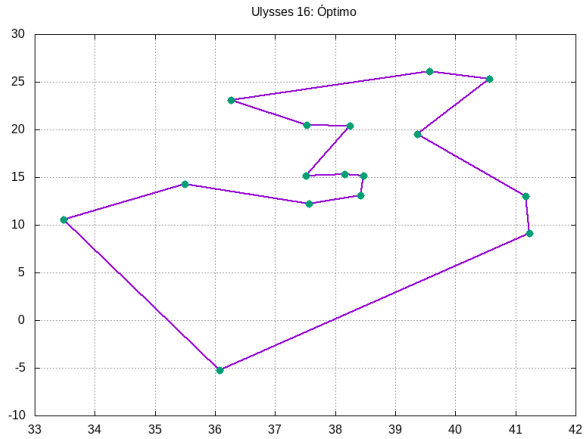
A280



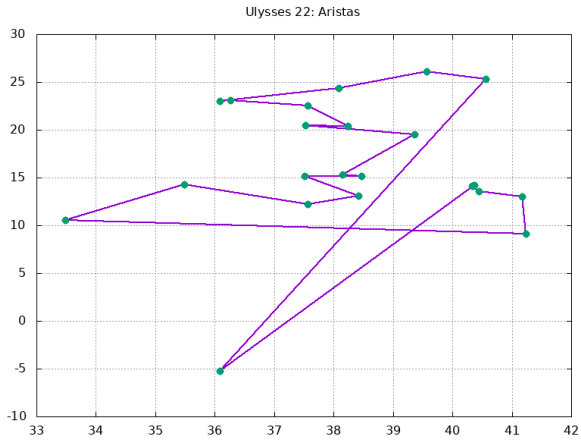


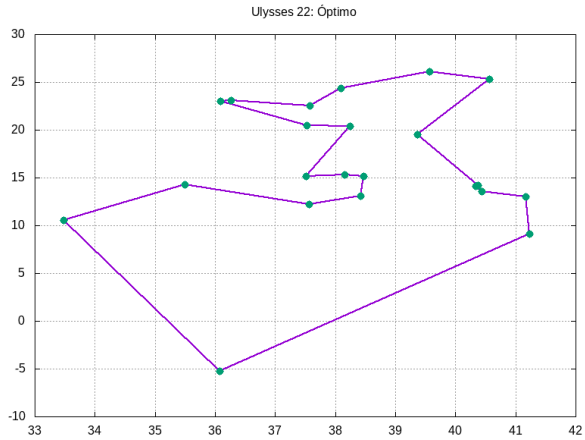
ATT48





Ulysses 16





Ulysses 22

TSP	Óptimo	Aristas	
		Camino	Diferencia
A280	2579	3199	24,04%
ATT48	33522	42155	25,75%
Ulysses 16	73	90	23,28%
Ulysses 22	74	91	22,97%

Se puede apreciar que la diferencia entre el resultado óptimo y el resuelto por aristas no varía demasiado entre los distintos resultados obtenidos.

2.4 Comparativa

Caminos Mínimos

TSP	Óptimo	Cercanía		Inserción		Aristas	
		Camino	Diferencia	Camino	Diferencia	Camino	Diferencia
A280	2579	3157	22,41%	4179	62,04%	3199	24,04%
ATT48	33522	40583	21,06%	55982	67,00%	42155	25,75%
Ulysses 16	73	103	41,1%	105	43,83%	90	23,28%
Ulysses 22	74	93	25,68%	107	44,59%	91	22,97%

Al haber programado distintas formas de resolver el problema TSP es fácil reconocer ahora que la forma seleccionada para resolverlo nos va a dar distintos resultados y soluciones a dicho problema.

En los escenarios A280 y ATT48 el resultado más próximo al óptimo se ha obtenido en el método de Cercanía. Sin embargo en los escenarios ulysses 16 y 22 se obtienen mejores resultados con nuestra solución propia. Por otro lado, el método de inserción obtiene caminos notablemente menos óptimos que los demás, lo cual se debe al escoger un recorrido inicial que recorre tres nodos bastante distantes entre sí

Es curioso observar como los algoritmos de inserción y el nuestro ofrecen unos resultados más similares; en tamaños más pequeños (16 y 22) ha funcionado mejor la resolución por Aristas, sin embargo, en tamaños mayores (48 y 280) el método por Cercanía ha sido más acertado, como hemos mencionado antes. Las diferencias de resultados obtenidos por estos dos algoritmos no han sido muy destacables. Cabe destacar también que el escenario de ejecución de ATT 48, Inserción ha obtenido una diferencia del 67%, la mayor diferencia obtenida.

Cabe destacar que los tres algoritmos son de eficiencia n^2 por lo qué sus variables ocultas son un valor a la hora de decidir cual de ellos usar si tenemos en cuenta su tiempo de ejecución.

3 Problema Asignación de Tareas

3.1 Descripción

En este problema se nos pide maximizar el beneficio que se puede generar realizando un conjunto de tareas. Si únicamente se tratase de maximizar el beneficio sin ninguna otra condición adicional, se realizarían todas las tareas sin importar el orden y sumaríamos todo el beneficio que se genera al realizar cada una de ellas. No obstante, se declara que cada tarea consume la misma cantidad de tiempo, una unidad, y que además, estas tienen un plazo limite para realizarse, por tanto, si superamos el plazo de una tarea, ya no podemos realizarla. Debemos por tanto encontrar una forma de realizar las tareas en un orden en el cual, podamos obtener el mayor beneficio posible.

Para encontrar una posible solución con enfoque voraz, vamos a ayudarnos del tipo de dato de la STL denominado como cola de prioridad (priority queue). La característica principal de este tipo de dato es que podemos ordenar en el mismo momento de introducción de los datos según el orden que definamos como prioridad. Además para no generar complicaciones en la lectura del código, se han encapsulado los datos de cada tarea en una clase. De esta forma, para poder determinar la prioridad de los objetos en la cola, se ha sobrecargado el operador <<.

3.2 Componentes del Algoritmo Voraz

- **Lista de candidatos:** Se puede observar claramente que tenemos un conjunto de candidatos pues son el conjunto de tareas que tenemos inicialmente.
- **Lista de candidatos utilizados:** El conjunto de candidatos ya usados son aquellos datos que han sido aceptadas para su ejecución y que inicialmente comienza vacío.
- **Función solución:** Realiza aquellas tareas que nos otorguen el beneficio máximo al ser ejecutadas.
- **Función selección:**La función de selección consiste en escoger la tarea que mayor beneficio otorgue.
- **Función de factibilidad:** La función de factibilidad es que no se puede realizar una actividad cuyo plazo límite ya haya sido superado.
- **Función objetivo:**La función objetivo es la misma que la función solución, es decir, que nuestro objetivo es encontrar un conjunto de tareas que den el beneficio máximo al ser ejecutadas.

3.3 Pseudocódigo

Para explicar el funcionamiento del código implementado, vamos a exponer el pseudocódigo del mismo.

```
funcion AsignacionTareas(priority_queue Q)
inicio
    TIEMPO_CONSUMIDO := 0
    BENEFICIO_TOTAL := 0
    lFinal := Cola vacia de tareas
    nTarea := Objeto tarea local

    mientras !Q.Vacio() hacer
        nTarea := Q.top()
        Q.pop()

        si TIEMPO_CONSUMIDO es menor o igual que tiempo de nTarea entonces
            BENEFICIO_TOTAL := BENEFICIO_TOTAL + nTarea.Beneficio()
            TIEMPO_CONSUMIDO := TIEMPO_CONSUMIDO + 1
            lFinal.insertar(nTarea)

        si no
            Se descarta la tarea
    fmientras
ffuncion
```

3.4 Proceso de generación de datos para el problema

Como para este problema no se proporcionan datos de análisis, se ha implementado una función auxiliar que genera los datos de las tareas cada vez que realizamos una ejecución nueva del problema. Este proporciona un número n de tareas que es pasado por parámetro.

-El identificador de la tarea se genera de forma aleatoria añadiendo una serie de números a la cadena "ID".

-A cada una de estas tareas se le asignará un plazo de tiempo cuyo rango va de 0 hasta el numero n de tareas.

-El beneficio de cada tarea se asigna de forma aleatoria y de rango 0-400.

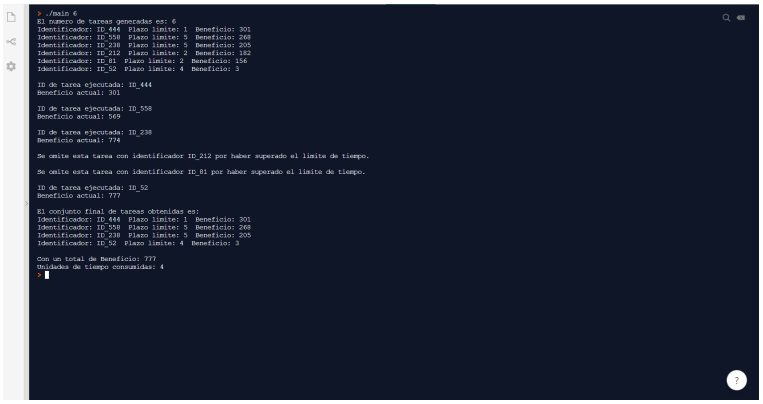
Cada tarea generada se irá añadiendo en la cola con prioridades, y la prioridad está marcada por el beneficio que tenga cada una. De esta forma, se mantendrá siempre en el tope de la cola aquella tarea que otorgue el mayor beneficio por ejecución.

3.5 Escenarios de ejecución

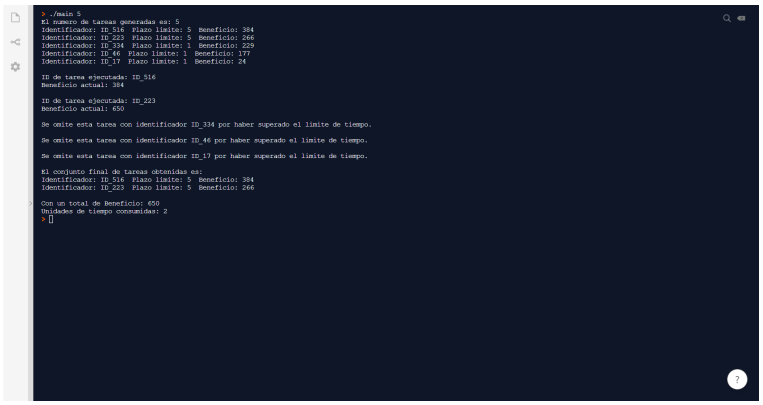
Para poder analizar los escenarios de ejecución del programa, se han realizado varias ejecuciones en los cuales se han obtenido diferentes resultados, que se van a explicar ahora.

3.5.1 Ejecución estándar

Los casos de ejecuciones estándar se dan cuando al formar el conjunto solución debemos descartar una o varias tareas ya que su plazo limite ha sido superado. Mostraremos un par de ejemplos sobre este caso y luego realizaremos una breve explicación



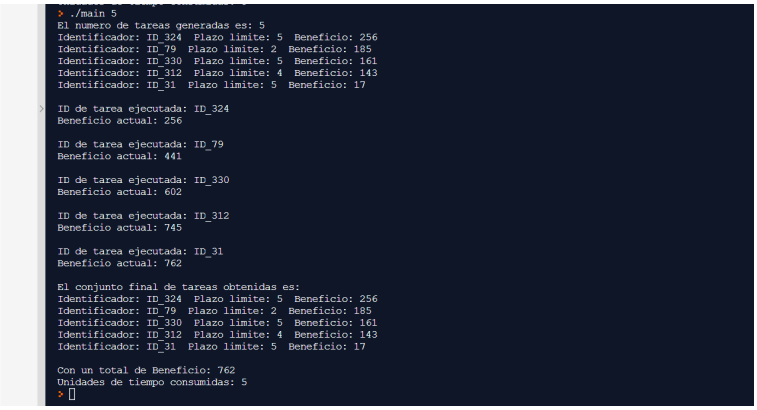
En este ejemplo vemos que, habiendo generado 6 tareas, hemos tenido que descartar 2, pues debido a la posición en la que se encuentran y habiendo consumido ya varias unidades de tiempo, el plazo límite de estos ya ha sido superado.



En este caso hemos tenido que descartar 3 tareas de las 5 que hemos generado, ya que hemos consumido dos unidades de tiempo al ejecutar las dos primeras tareas y para las tres restantes, el plazo límite es 1, por eso debemos descartar las tareas restantes.

3.5.2 Ejecución ideal

Ahora vamos a ver un caso de ejecución ideal. Este se daría si se pueden realizar todas las tareas que tenemos en el conjunto inicial. Esto depende tanto del beneficio que generen cada una de las tareas como de los plazos límites que tengan cada una de ellas



En este caso podemos observar que como los límites de plazo son altos (el único límite de plazo medianamente bajo es la segunda tarea) se pueden realizar todas las tareas sin ningún inconveniente.

4 Conclusión

A la vista de lo estudiado, se puede concluir que los algoritmos Greedy o voraces, si bien en varios casos este tipo de algoritmos no logran obtener un resultado óptimo, esto no quiere decir que no tengan utilidad, se sabe que con problemas que son del tipo NP-Completos una aproximación basada en Greedy puede dar, dependiendo de la heurística a utilizarse, unos resultados muy buenos comparados a lo que daría un algoritmo que diese el óptimo. Teniendo en consideración que estos algoritmos pueden poseer una complejidad exponencial cuando se trata de problemas como el vendedor viajante, realizar una aproximación que dé un resultado cercano pero con complejidad polinómica es un sacrificio que, dependiendo a lo que se aplique, vale la pena realizar, esto sumado a que estos algoritmos son relativamente sencillos de implementar, demuestran que tienen aplicaciones en cualquier índole de la informática.