Algorítmica - Practica 3: Algoritmos Voraces

A. Herrera, A. Moya, I. Sevillano, J.L. Suarez

12 de mayo de 2015

Contents

1 Organización de la práctica

La práctica 3 trata sobre el desarrollo de algoritmos greedy que consigan la solución óptima de los problemas propuestos o actúen como heurística sobre los mismos. Uno de los problemas a estudiar es el viajante de comercio, ampliamente conocido en el ámbito de la inteligencia artificial y teoría de algoritmos. Es un problema NP completo pero que será abordable gracias al uso de algoritmos greedy polinomiales que no proporcionarán la solución óptima pero sí una lo suficientemente buena para nuestros objetivos.

Nuestro grupo debe resolver el problema 4 y el viajante de comercio. Hemos abordado también el problema opcional (número 5) obteniendo el algoritmo greedy óptimo.

Para cada problema se sigue la siguiente estructura:

- Enunciado del problema
- Resolución teórica del problema (con una subsección por algoritmo)
- Análisis empírico. Análisis de la eficiencia híbrida

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la eficiencia híbrida para los mismos.

Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Toshiba
- **RAM:** 8 GB
- **Procesador:** Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz

El código, los resultados de las ejecuciones, las gráficas y los pdf asociados se pueden encontrar en GitHub.

2 Problema 4

2.1 Enunciado del problema

Consideremos un grafo no dirigido $G = (V, E)$. Un conjunto U se dice que es un recubrimiento de G si $U \subseteq V$ y cada arista en E incide en, al menos, un vértice o nodo de U, es decir,

$$\forall (x, y) \in E : x \in U \text{ o } y \in U$$

Un conjunto de nodos es un recubrimiento minimal de G si es un recubrimiento con el menor número posible de nodos.

- Diseñar un algoritmo greedy para intentar obtener un recubrimiento minimal de G. Demostrar que el algoritmo es correcto, o dar un contraejemplo.
- Diseñar un algoritmo greedy que obtenga un recubrimiento minimal para el caso particular de grafos que sean árboles.
- Opcionalmente, realizar un estudio experimental de las diferencias entre los dos algoritmos anteriores cuando ambos se aplican a árboles.

2.2 Solución teórica

En este apartado exponemos la solución teórica del problema para grafos arbitrarios y para árboles. Los algoritmos se ejemplificarán sobre el árbol de la Imagen 1.

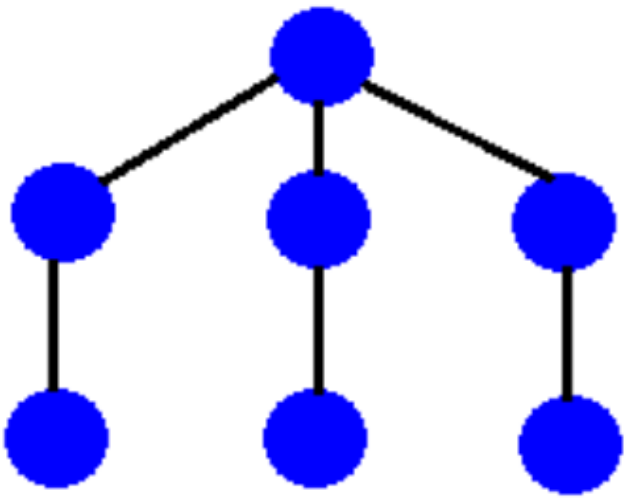


Imagen 1. Árbol sobre el que aplicaremos los algoritmos.

El recubrimiento minimal del árbol de la Imagen 1 viene dado por los nodos en rojo de la Imagen 2.

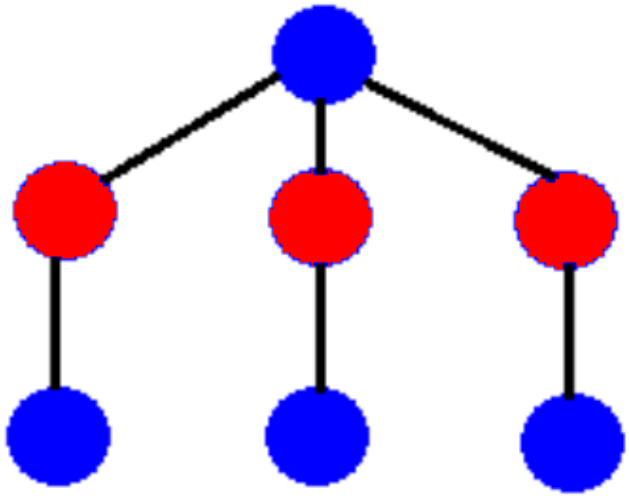


Imagen 2. Recubrimiento minimal para el árbol de la Imagen 1.

2.2.1 Solución para un grafo arbitrario

Tras múltiples intentos que se narrarán a continuación, no conseguimos un algoritmo polinomial óptimo para el problema sobre grafos arbitrarios por lo que pensamos que este en su versión de decisión era NP. De hecho, no solo es NP sino que es NP Completo. Más información al respecto se puede encontrar en el siguiente enlace. Procedemos a explicar los algoritmos greedy desarrollados para conseguir una aproximación aceptable del problema.

En primer lugar, nótese que si tomamos $U = G$ tenemos un recubrimiento. Este será el peor recubrimiento posible pues utiliza todos los nodos del grafo. Queremos obtener un mejor recubrimiento. Para ello, construyamos uno desde 0:

Algoritmo 1. Un primer intento de algoritmo aleatorio.

1. $U = \emptyset$
2. Para cada arista $(x, y) \in E$ tomamos como nodo x o y aleatoriamente y lo añadimos a U

Este algoritmo aleatorio obtiene un recubrimiento del grafo pues para cada arista hay efectivamente un nodo en U sobre el que esta incide. El tiempo de ejecución es lineal sobre el número de aristas, $\theta(|E|)$. Sin embargo, es claro que normalmente las soluciones obtenidas serán manifiestamente mejorables. Podemos mantener este tipo de construcción pero librarnos parcialmente de la aleatoriedad introduciendo una estrategia greedy al algoritmo:

Algoritmo 2. Mejora del Algoritmo 1 siguiendo una estrategi voraz.

1. $U = \emptyset$
2. Para cada arista $(x, y) \in E$ tomamos un nodo v y lo añadimos a U donde v es:
   - x si $x \in U$.
   - y si $y \in U$.
   - Uno de los dos, elegido aleatoriamente, si $x, y \notin U$.

En efecto, si algún nodo sobre el que incide la arista ya está en U no tenemos por qué añadir uno nuevo.

Sin embargo, este algoritmo greedy, lineal sobre el número de aristas y aleatorio no es óptimo como cabe esperar. No es difícil encontrar un ejemplo en el que la aleatoriedad provoque una mala solución. Podemos considerar el árbol dado en la Imagen 1 y observar que es factible que una ejecución del algoritmo obtenga tanto el óptimo dado en la Imagen 2 como el resultado mostrado en la Imagen 3.

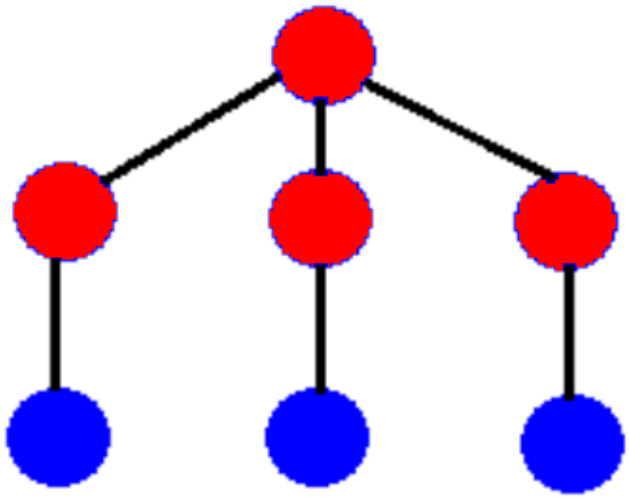


Imagen 3. Un recubrimiento no minimal del árbol de la Imagen 1.

Nuestro siguiente intento de algoritmo voraz consiste en construir la solución desde otra perspectiva. En lugar de añadir un nodo por arista elegido bajo cierto criterio voraz añadimos nodos de forma genérica hasta obtener un recubrimiento.

Consideramos para cada nodo su grado. El grado de un nodo es el número de aristas que inciden en él. Es lógico tomar en primer lugar el nodo con mayor grado ya que así no tenemos que preocuparnos de añadir un nodo para las aristas que inciden sobre él. Podemos abstraer este criterio como una función de selección para nuestro algoritmo greedy. En cada iteración añadimos a U el nodo con mayor grado hasta que toda arista incida sobre algún nodo de U. Sin embargo, puede darse el caso de que estemos introduciendo un nodo innecesario pues las aristas que inciden en él ya inciden sobre algún otro nodo de U. Decidimos ante esta situación realizar la siguiente operación: cada vez que un nodo se añade a U se elimina el nodo de V y se eliminan de E las aristas que inciden sobre él. Posteriormente recalculamos el grado de cada nodo para este nuevo grafo. Nuestro algoritmo quedaría de la siguiente forma:

Algoritmo 3. Algoritmo greedy basado en grados.

```
# G = (V,E) es el grafo sobre el que se ejecuta el algoritmo
U = []
while not E.isEmpty:
    v = V.nodeMaximumDegree()
    U.add(v)
    for edge in E:
        E.delete(edge) if edge[0] == v or edge[1] == v
    V.delete(v)
```

Sin embargo, este algoritmo no es óptimo. Un ejemplo de este hecho puede ser la Imagen 3. El recubrimiento mostrado es precisamente el recubrimiento dado por este algoritmo.

2.2.2 Solución para un árbol

Hemos visto que todos los algoritmos anteriores podrían fallar incluso en un árbol. Sin embargo, estos algoritmos genéricos para grafos no aprovechaban este hecho. En un árbol se pueden deducir fácilmente propiedades sobre su recubrimiento minimal.

Proposición 1.
Sea $T = (V, E)$ un árbol. Entonces, existe un recubrimiento minimal del mismo en el cual no contiene a ninguna hoja del árbol pero sí a todo padre de una hoja.

Demostración. Consideremos $U \subset V$ un recubrimiento minimal de T. Si ninguna hoja del árbol está en U se tiene el resultado. En caso contrario, para cada hoja del árbol en U añadimos su padre y la eliminamos obteniendo así el recubrimiento U'. El número de nodos en U' es el mismo que el de U. Es por tanto un recubrimiento minimal de T sin ninguna hoja. Además, se debe tener que contiene a todo padre de una hoja por el hecho de que la arista que los une tiene al menos un nodo en U'.

■

Vamos a calcular un recubrimiento minimal para un árbol T. Para ello usaremos la proposición 1, que nos da información sobre un posible recubrimiento minimal. Se propone el siguiente algoritmo:

Algoritmo 4. Algoritmo óptimo para árboles.

```
# T es el árbol sobre el que se ejecuta el algoritmo.
# Se asume que se ha tomado una raíz para T.
# hojasUltimoNivel() calcula las hojas del árbol
# que se encuentran en el último nivel de este.
U = []
while not T.isEmpty:
    hojas_ultimo_nivel = T.hojasUltimoNivel()
    for hoja in hojas_ultimo_nivel:
        U.append(hoja.parent)
        T.delete([hoja, parent])
```

Su funcionamiento es el siguiente, se recorre el árbol por niveles de abajo hacia arriba. En cada iteración se calculan las hojas del árbol en el nivel correspondiente y se añaden los padres al futuro recubrimiento, siguiendo la filosofía de la proposición 1. Posteriormente se eliminan las hojas, sus padres y las aristas que inciden en estos de T y se repite el proceso.

Proposición 2.
El algoritmo 4 calcula el recubrimiento minimal del árbol.

Demostración. En primer lugar, es fácil darse cuenta que tras la última iteración, U es un recubrimiento de T. Veamos que es minimal. Basta ver que existe un recubrimiento minimal de T que contiene a U. Este hecho se prueba por inducción sobre las iteraciones del algoritmo. Denotamos T_i al grafo al inicio de cada iteración.

- Para la iteración 1, basta tomar como recubrimiento minimal de T el dado por la proposición 1. Además, T_2 resultado de eliminar a T_1 las hojas en el último nivel y sus padres es un árbol.
- Supongamos el resultado cierto para la iteración $i-1$. Esto es, T_i es un árbol y existe W, recubrimiento minimal de T que contiene a U. Veamos que tras realizar la iteración sigue existiendo tal recubrimiento. U contiene ahora a los padres de las hojas del último nivel de T_i. Se tiene que $W - U$ es un recubrimiento minimal de T_i por reducción al absurdo. Si no lo fuese, tomamos W_i recubriento minimal de T_i dado por la proposición 1. $W' = W_i \cup U$ es un recubrimiento de T con menos nodos que W, contradicción. Por tanto, $W - U$ tiene el mismo número de nodos que W_i. Esto implica que $W' = W_i \cup U$ tiene el

mismo número de nodos que W, luego es un recubrimiento minimal de T que contiene a U como se quería. Además, si tomamos T_{i+1} resultado de quitar las hojas del último nivel y los padres de estas a T_i sigue siendo un árbol.

■

El algoritmo es en esencia greedy pues en cada iteración realizamos una elección de elementos del recubrimiento U bajo nuestro propio criterio (padre de una hoja del último nivel), que resulta dar el recubrimiento óptimo. Una mejor implementación del algoritmo se puede lograr usando el recorrido en post-orden del árbol como se hace en el algoritmo 5.

Algoritmo 5. Algoritmo óptimo para árboles con recorrido en post-orden.

```
# Si T es el árbol sobre el que se desea aplicar el algoritmo,
# realizar algoritmoOptimo(T.raiz).
# Parámetros: v es un nodo del árbol
def algoritmoOptimo(v):
    U = []
    # Se calcula primero U para el nivel inferior.
    for hijo in v.hijos:
        U = U.union(algoritmoOptimo(hijo))
    # Si algún hijo no está en U, se añade v.
    for hijo in v.hijos:
        if hijo not in U:
            U.append(hijo); break

    return U
```

Proposición 3.
El algoritmo 5 obtiene el mismo recubrimiento que el algoritmo 4.

Demostración. Este hecho se puede probar por inducción sobre los niveles del árbol.

- El caso base es cuando v es una hoja, que no tiene hijos y por ello no se añade a U. Además, su padre sí se añade posteriormente como pasa en el algoritmo 4.
- Si v tiene hijos, supongamos como hìpótesis de inducción que para cada subárbol que cuelga de estos el resultado es el mismo que el que daría el algoritmo 4. Se toma U como la unión de tales resultados. Si todos los hijos de v están en U, el algoritmo 4 no escogería a v pues sería una hoja en determinada iteración. En caso contrario, un hijo de v sería una hoja en alguna iteración al no estar nunca en U y v se añade a U. En cualquier caso, se obtiene el mismo resultado que el algoritmo 5.

■

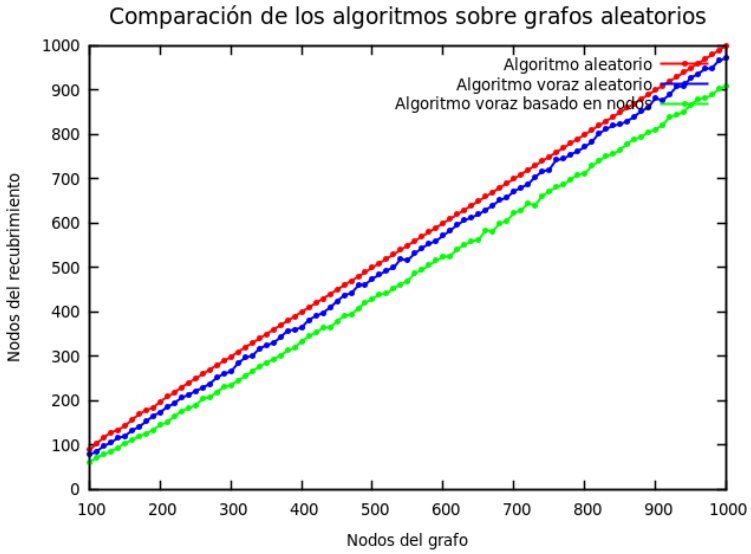
La implementación del algoritmo 5 tiene la misma eficiencia que un recorrido en post-orden, esto es, lineal sobre el número de nodos. Nótese que al aplicarlo sobre el árbol de la Imagen 1 se obtendría el resultado dado en la Imagen 2.

2.3 Resultados obtenidos experimentalmente.

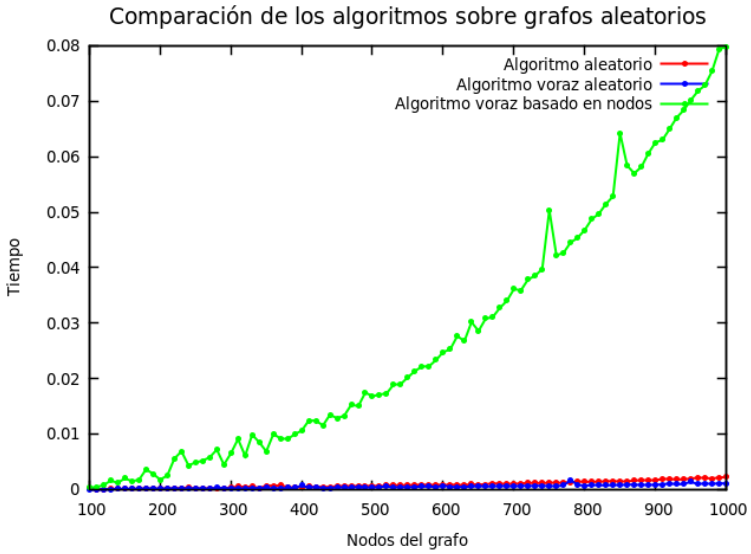
Para probar cada uno de los algorímos experimentalmente, y que todos ellos se ejecutasen sobre el mismo grafo(o arbol) hemos implementado dos generadores tanto de arbol como de grafos aleatórios. En el primero usámos, además de el número de nodos para generar el grafo, el concepto de densidad de grafo para generarlos con más o menos aristas. Este hecho influye en el tiempo de ejecución de los algorítmos. En el segundo usamos el número de hijos máximo, hecho que tambien lo hace. No incluiremos el estudio de el cámbio en tiempo de ejecución con respecto a estos parámetros, aunque si lo notaremos como anécdota.

2.3.1 Eficiencia en número de nodos en grafos.

Los tres algorítmos utilizados para grafos dan resultados bastante parecidos, aunque siempre ha de tenerse en cuenta que conforme se mejora la idea, se eliminan nodos de el recubrimiento. Así queda la gráfica:

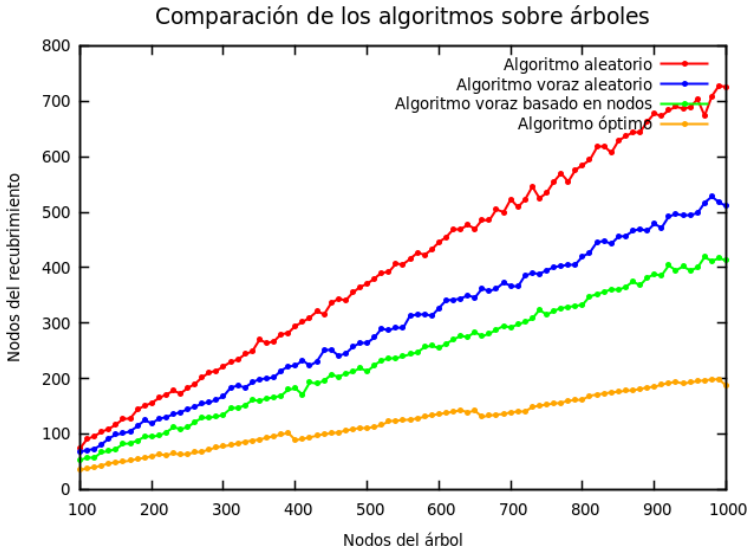


Sin embargo, si nos fijamos en la gráfica de tiempos, vemos que el comportamiento del tercer algorítmo es bastante más costoso, $O(n^2)$, por lo que nos planteamos si merece la pena conseguir un recubrimiento insignificantemente mejor por un tiempo cuadrático:

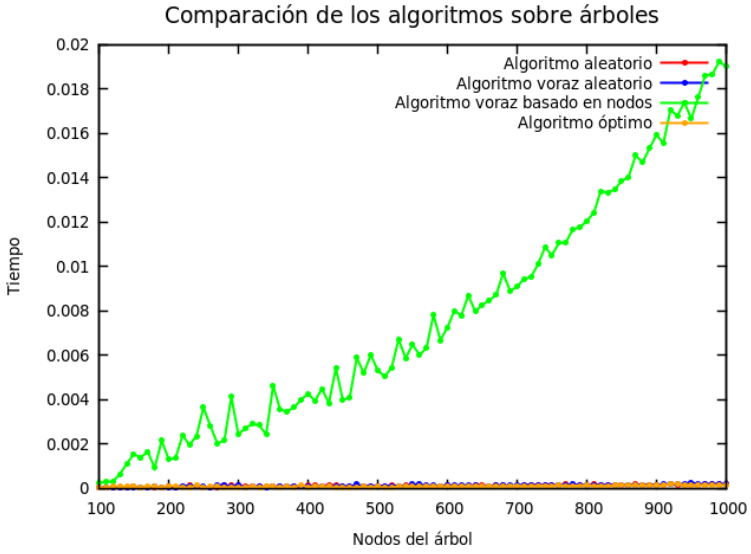


2.3.2 Eficiencia en número de nodos en aroles.

Los resultados de ejecutar los algorítmos anteriores y el algorítmo creado para el caso de que el grafo sea un arbol. Los resultados son notáblemente mejores en cuanto alrecubriemiento minimal que se obtiene en este último:



Además de ser notablemente mejor en cuanto a número de nodos, el tiempo, como ya demostramos, tiene un coste en tiempo com paráble a los algorítmos priemero y segundo para grafos, que son lineales:



3 Problema 5

3.1 Enunciado del problema

Un electricista necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente y esta es inversamente proporcional al tiempo que tardan en atenderles, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de atención de los clientes (desde el inicio hasta que su reparación es efectuada).

- Diseñar un algoritmo greedy para resolver esta tarea. Demostrar que el algoritmo obtiene la solución óptima.
- Modificar el algoritmo anterior para el caso de una empresa en la que se disponga de los servicios de más de un electricista.

3.2 Solución teórica

Dados $\{t_i : i = 1, \ldots, n\}$, buscamos una permutación $x = (i_1, \ldots, i_n)$ de los n primeros números naturales que minimice

$$f(x) = \frac{1}{n} \sum_{k=1}^{n} T_k \text{ donde } T_k = \sum_{j=1}^{k} t_{i_j}$$

T_k es el tiempo de espera del cliente i_k. Se explican a continuación las soluciones para uno o más electricistas.

3.2.1 Algoritmo greedy óptimo para un único electricista

Desarrollemos un algoritmo greedy para el problema. En cada momento, el electricista elige un trabajo de los que le quedan pendientes y lo realiza. Parece lógico elegir aquel que más corto va a ser, pues si elegimos uno de mayor duración mucha gente tendrá que esperar a su finalización, aumentando el tiempo medio de espera. Así pues, proponemos el algoritmo 1.

Algoritmo 1. Algoritmo greedy para un electricista.

```
# t es el vector con los tiempos de los trabajos
sol = []
for i in range(0, n):
    siguiente_trabajo = t.index(t.min())
    sol.append(siguiente_trabajo)
    t[siguiente_trabajo] = math.inf
```

O, equivalentemente,

1. Ordenar los tiempos $\{t_i : i = 1, \ldots, n\}$ de menor a mayor.
2. Tomar como solución la permutación que los ordena $(i_1, \ldots, i_n)$. Esto es equivalente a tomar como solución $(1, \ldots, n)$ para los tiempos ordenados.

Este algoritmo tan sencillo de eficiencia $\theta(n \log n)$, es efectivamente el óptimo para el problema.

Proposición 1.
El algoritmo 1 minimiza el tiempo medio de espera.

Demostración. Ordenamos el vector con los tiempos de menor a mayor. La solución dada por el algoritmo 1 para el vector de tiempos ordenado es $x = (1, \ldots, n)$. Veamos que cualquier otra permutación $x' = (i_1, \ldots, i_n)$ tiene mayor o igual tiempo medio de espera. Tomamos el primer índice j tal que $t_{i_j} > t_{i_{j+1}}$. Si este índice no exíste, entonces el tiempo medio de x' es el mismo de x. En caso de que exista, transponemos i_j con i_{j+1}. Esta nueva permutación x'' tiene menor tiempo medio de espera que x':

$$f(x') - f(x'') = \frac{1}{n}(t_{i_j} - t_{i_{j+1}}) > 0$$

Tomamos la nueva permutación como x'. Podemos repetir el proceso hasta que no exista j. Por la transitividad del orden, la primera permutación x' tiene mayor tiempo medio de espera que la final, cuyo tiempo medio de espera es el de x.

■

Nótese que la prueba es básicamente un algoritmo de ordenación burbuja.

3.2.2 Algoritmo greedy óptimo para varios electricistas

Mantenemos la idea anterior para este caso. Cada vez que un electricista termina un trabajo elige el siguiente por hacer que menos tiempo requiera. El orden en el que se efectuarán los trabajos es el mismo que en el apartado anterior. Cambia el tiempo medio de espera pues varios trabajos se ejecutan en paralelo.

4 Problema del TSP

4.1 Enunciado del problema

- Implementar un programa que proporcione soluciones para el problema del viajante del comercio empleando las heurísticas del vecino más cercano y la mejor inserción, así como otra adicional propuesta por el propio equipo. El programa debe proporcionar el recorrido obtenido y la longitud de dicho recorrido.
- Realizar un estudio comparativo de las tres estrategias empleando un conjunto de datos de prueba.

4.2 Heurística del Vecino más Cercano

Comenzaremos con la estrategia del vecino más cercano. Para ello, identificaremos el algoritmo de resolución como un algoritmo greedy. Para darle este enfoque debemos identificar primero las características que lo definen:

- **El conjunto de candidatos.** Claramente, este conjunto es el que forman todas las ciudades de nuestro problema a resolver.
- **Candidatos usados.** Este conjunto lo constituyen las ciudades con las que vamos construyendo nuestra solución parcial, es decir, las ciudades ya visitadas.
- **La función solución.** Para los problemas del viajante de comercio, la función solución simplemente tiene que verificar si el número de ciudades visitadas coincide con el total de ciudades que componen el problema.
- **La función de selección.** Al elegir la heurística del vecino más cercano, la función de selección se convierte en aquella que, para cada ciudad C, se devuelve aquella ciudad S del conjunto de candidatos no usados E, tal que $dist(S, C) = \min\{dist(P, C) : P \in E\}$.
- **La función objetivo.** Esta función es, como para cualquier problema del viajante de comercio, la distancia total del recorrido, que es la que tratamos de optimizar.

De esta forma, una vez captada la esencia greedy de nuestra heurística, podemos plantear de forma sencilla el pseudocódigo para nuestro algoritmo:

```
def TSPVecinoMasCercano
  solucion = []
  solucion.push(conjuntoCiudades[0])  #Comenzamos añadiendo una ciudad inicial sobre la
                                      # que se empiezan a buscar los vecinos
                                      # (puede ser cualquiera)

  candidatos = conjuntoCiudades
  candidatos.erase(cojuntoCiudades[0])

  while solucion.numeroCiudades < totalCiudades # (función solución)

    # Vamos buscando el vecino más cercano a la última ciudad insertada,
    # con nuestra función de selección previamente indicada.
    C = candidatos.vecinoMasCercano(solucion.get(solucion.numCiudades-1))
    solucion.push(C)
    candidatos.erase(C)
  end
  return solucion
end
```

4.3 Heurística de la mejor inserción

Continuamos con la estrategia de la mejor inserción. De nuevo vamos a tratar de darle un enfoque greedy, identificando sus distintas propiedades, como se hizo anteriormente. Tenemos que el conjunto de candidatos, los candidatos usados, la función solución y la función objetivo coinciden con el caso anterior, y lo harán igualmente con cualquier algoritmo que trate de resolver el problema del viajante de comercio. Nos queda por identificar la función de selección, que es la que caracteriza a la heurística para resolver el problema del viajante de comercio.

Función de selección. La función de selección en este caso será aquella que, dada una solución parcial $\{S_i\}_{,i=1,...,n}$, obtiene una nueva ciudad del conjunto de candidatos no usados, $C \in E$, y una permutación π_j que inserta C en una posición j de $\{S_i\}$. Esto es, considerando $C := S_{n+1}$ la permutación viene dada por

$$\pi_j : \{1, ..., n+1\} \to \{1, ..., n+1\}$$

$$\pi_j(k) = k \ \forall k < j, \pi_j(n+1) = j, \text{ y } \pi_j(k) = k+1 \ \forall j < k < n+1)$$

de forma que $\{S_{\pi_j(i)}\}_{i=1,...,n+1}$ minimiza la distancia del recorrido parcial, esto es, la función objetivo. Esta función puede resultar extraña a primera vista, pero la idea del algoritmo es intuitiva. Se resume en ir construyendo a partir de cada recorrido parcial de forma inductiva, un nuevo recorrido añadiendo una ciudad válida que minimice el recorrido al ser insertada en una posición adecuada.

Una vez comprendida nuestra nueva heurística, ya podemos desarrollar un pseudocódigo para nuestro algoritmo. Solo falta definir un patrón de comienzo. Lo hacemos esto de forma sencilla. Empezamos por una ciudad i, y sobre ella iremos realizando inserciones. Podemos incluso posteriormente realizar este proceso comenzando por cada una de las ciudades, y quedarnos con el mejor resultado obtenido.

```
def TSPMejorInsercion
  solucion = []
  solucion.push(conjuntoCiudades[i])  # Comenzamos añadiendo una ciudad inicial sobre la
                                      # que se buscarán las mejores inserciones.

  candidatos = conjuntoCiudades
  candidatos.erase(cojuntoCiudades[i])

  while solucion.numeroCiudades < totalCiudades # (función solución)

    # Buscamos la ciudad y la posición en la que insertarla según la heurística de la
    # mejor inserción. Esta es nuestra función de selección.
    [C,posicion] = candidatos.mejorInsercion(solucion)
    solucion.insert(C,posicion)
    candidatos.erase(C)
  end
  return solucion
end
```

4.4 Heurística propuesta: Optimización de la Colonia de Hormigas

Finalmente, propondremos una nueva heurística bioinspirada para resolver el problema del viajante de comercio, que funciona cierto grado de aleatoriedad y con un aire de similitud a un algoritmo voraz. Para ello nos basaremos en estrategias que ha proporcionado la propia naturaleza en determinadas situaciones para resolver problemas equivalentes. Nos centraremos concretamente en la actuación de las hormigas.

En primer lugar, un algoritmo bioinspirado es aquel que se basa en la naturaleza, imitando su comportamiento para encontrar la solución a un problema. Su desarrollo ha crecido en los ultimos años, dando lugar a

importantes presentantes como los algoritmos genéticos o la optimización de la colonia de hormigas, que nos ocupa ahora.

Muchas veces hemos tenido la oportunidad de observar a las hormigas saliendo en grupo para buscar alimentos. Una gran cantidad de estos pequeños insectos, juntos, unos detrás de otros, formando un camino que se aprecia claramente a simple vista. Lo que seguramente no hemos apreciado nunca es que las hormigas, además de formar estas rutas, son capaces de transformarlas con el tiempo en un camino mínimo entre su lugar de salida y su objetivo. Maravillosamente, las hormigas conocen desde tiempos inmemoriables secretos del TSP que tantos quebraderos de cabeza han podido causar a las personas en los últimos tiempos. ¿Cómo pueden estos minúsculos seres realizar esta impresionante hazaña? Lo veremos a continuación.

Resulta que las hormigas son ciegas. Para desplazarse, se guían a traves del olfato, concretamente, a través de las feromonas que van soltando, tanto ellas como otras hormigas. Cada vez que una hormiga sale de viaje, va dejando un rastro de feromonas que luego sigue para volver a su hormiguero. Debemos tener en cuenta también que los rastros de feromonas no son eternos; se van evaporando con el paso del tiempo. Además, los rastros de feromonas no son un camino a seguir por fuerza; una hormiga despistada puede crear una nueva ruta a su antojo, abriendo posibilidades a nuevas rutas. Ahora bien, si dos hormigas realizan recorridos distintos entre un mismo origen y destino, ¿qué puede ocurrir?

Supongamos, en la situación anterior, que tenemos dos hormigas A y B, y que la ruta que ha encontrado A es más corta que la de B. Esto quiere decir que A irá y volverá en menos tiempo, dejando un rastro de feromonas a la ida y reforzándolo a la vuelta, por lo que la probabilidad futuras hormigas sigan su mismo camino aumentará. Por el contrario, B tardará más tiempo en ir y volver, por lo que habrá más evaporación de feromonas y su rastro tendrá menos intensidad; cada vez será seguido por menos hormigas y terminará por desaparecer. Esta situación se repetirá cada vez que una hormiga trate de buscar un nuevo camino pasando del rastro de feromonas. Si el camino es mejor que los anteriores, acabará por atraer a más hormigas y se convertirá en un camino dominante. En cambio, si el camino no es bueno, su ruta acabará por desaparecer. Es, de esta forma, como las hormigas siempre consiguen hallar una ruta que seguir entre su origen y su destino, y no solo una ruta cualquiera, si no una buena ruta, en términos de distancias. De esta maravilla de la naturaleza extraeremos nuestra nueva heurística para el problema del viajante de comercio.

Una vez planteada nuestra idea, veamos que realmente se trata de un algoritmo voraz. Como en los demás algoritmos para el problema del viajante de comercio, es claro cuáles son los candidatos, la función solución y la función objetivo. La función de selección se corresponde con la forma en la que eligen las hormigas el camino a seguir (a qué ciudad se mueven desde el punto en el que están), y está basado en ciertas leyes probabilísticas que se deducen de la cantidad de feromonas que haya en el camino entre cada dos ciudades.

No tenemos una función de selección propiamente dicha, ya que puede tomar valores distintos para una misma entrada, es decir, tiene cierto grado de aleatoriedad. Estamos ante un algoritmo voraz aleatorizado. Se pueden encontrar otros algoritmos voraces aleatorizados en la literatura especializada, como los utilizados para el funcionamiento del algoritmo GRASP. Además, esta fución de selección aleatoria evoluciona con el tiempo juto con la distribución de feromonas en el grafo. Dicha distribución de feromonas se representa mediante el concepto de *mapa de feromonas*. Su implementación se realiza tomando una matriz isomorfa a la matriz de distancias entre ciudades, que represente al grafo ponderado con la cantidad de feromonas.

Por último, al igual que las hormigas requieren de muchos paseos para obtener un camino óptimo, será necesario dejar iterar el algoritmo cierto número de veces para que las soluciones obtenidas vayan mejorando a la vez que evoluciona el mapa de feromonas.

Un pseudocódigo para nuestra nueva heurística, sin entrar en detalle en las leyes de probabilidad por las que puede regirse el algoritmo, es el siguiente:

```
def TSPAntsColonyOptimization(numIteraciones)
  solucion = []

  while i < numIteraciones
    # Creamos una colonia de hormigas (de tamaño a elegir) y las soltamos en
```

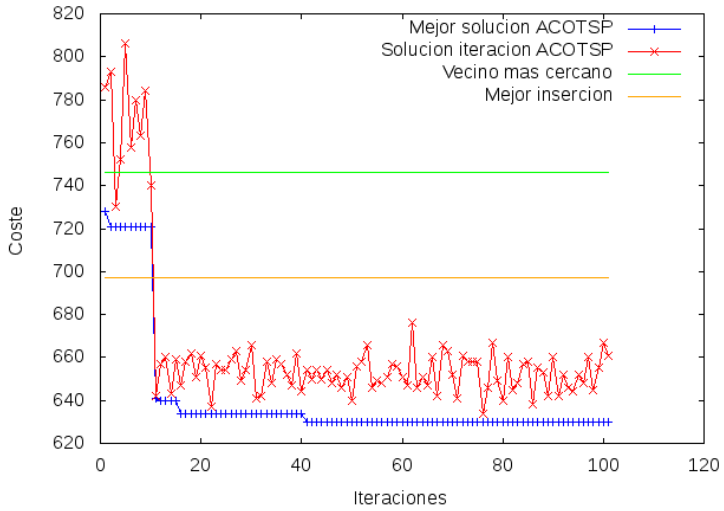
```
    # una ciudad al azar.
    # Las hormigas forman una estructura que se mueve a la vez sobre el mapa de ciudades
    # y el mapa isomorofo de feromonas y funcionará como un agente que proporciona
    # diversas funcionalidades:
    # - Memoria de recorrido: almacenan su ruta y permiten recuperar las soluciones que
    #   obtienen.
    # - Movimiento probabilístico: son capaces de determinar la ciudad a la que van a
    #   avanzar teniendo en cuenta probabilidades dadas por la cantidad de feromonas que
    #   encuentran en cada ruta.
    # - Actualización de feromonas: con cada movimiento, las hormigas actualizan el mapa
    #   de feromonas con su aportación.

    for hormiga in coloniaHormigas
      hormiga.iniciaRuta(rand)
    end

    # Hacemos que las hormigas hagan un camino completo para obtener así una solución.
    for i in 1..numeroCiudades
      for hormiga in coloniaHormigas
        C = hormiga.determinarSiguienteCiudad() #Función de selección probabilística.
        hormiga.avanza(C) #La hormiga avanza en su recorrido y añade feromonas.
      end
    end

    # Una vez las hormigas han obtenido la solución, nos vamos quedando con la mejor.
    for hormiga in coloniaHormigas
      solucionHormiga = hormiga.getSolucion
      if solucion == [] or solucionHormiga.coste() < solucion.coste()
        solucion = solucionHormiga
      end
    end
    # Repetimos el proceso el número de iteraciones que se pide en el argumento. Las
    # feromonas se irán modificando y dando lugar cada vez a soluciones mejores.
  end
end
```

El algoritmo que obtenemos es más pesado que los vistos anteriormente. Requiere mayor coste y es menos eficiente, pues necesita bastantes iteraciones, y en cada iteración muchos recorridos, mientras que los otros solo construyen una única solución. Aun así, supera con creces la eficiencia del algoritmo de fuerza bruta, que es factorial. Además, se tiene la ventaja de que, mientras que los greedy anteriores obtienen siempre la misma solución, este algoritmo obtiene mejores soluciones conforme aumentan las iteraciones, mejora dada por el grado de aleatoriedad al que está sometido. De todas formas es posible mejorarlo introduciendo el concepto de mejora local de una solución.

Una búsqueda local de una solución es un nuevo tipo de heurística que parte de una solución existente, que se puede construir mediante una heurística sencilla, y obtiene una nueva solución basada en la anterior mejorándola todo lo posible según las restricciones de la heurística de mejora. Este tipo de mejoras se reduce, básicamente, a tomar una vecindad del conjunto de soluciones para la solución existente, y dentro de esta vecindad, elegir una solución mejor que la actual. De esta forma vamos obteniendo óptimos locales, si bien las soluciones optenidas pueden no ser la óptima para el problema, sí son muy buenas.

La idea de la búsqueda local se puede aplicar a las hormigas de tal forma que, tras obtener una solución por parte de cada hormiga, estas soluciones pueden ser mejoradas localmente, y si aplicamos el cambio de feromonas sobre las soluciones mejoradas obtenemos una convergencia mucho más rápida y soluciones muy buenas. Al combinar las búsquedas locales con el algoritmo de optimización de las hormigas, obtenemos

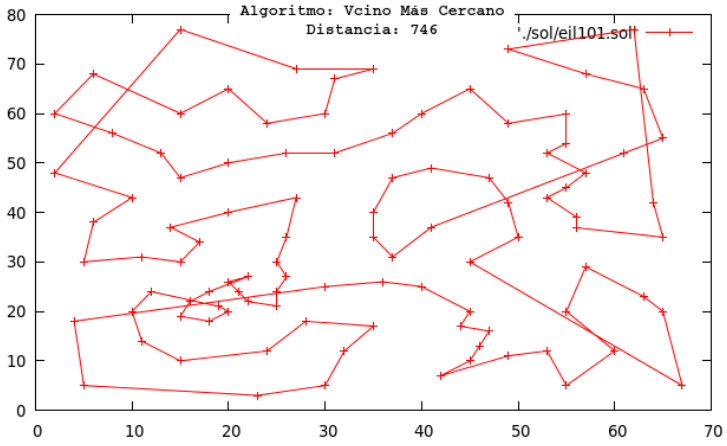
un algoritmo que sigue siendo relativamente pesado en comparación con los de vecino más cercano y mejor inserción, pero abordable en tiempo polinómico. La velocidad de mejora de las soluciones se muestra en la siguiente gráfica, con la implementación del algoritmo en C++, que se puede consultar en el siguiente enlace:



4.5 Comparación de las heurísticas

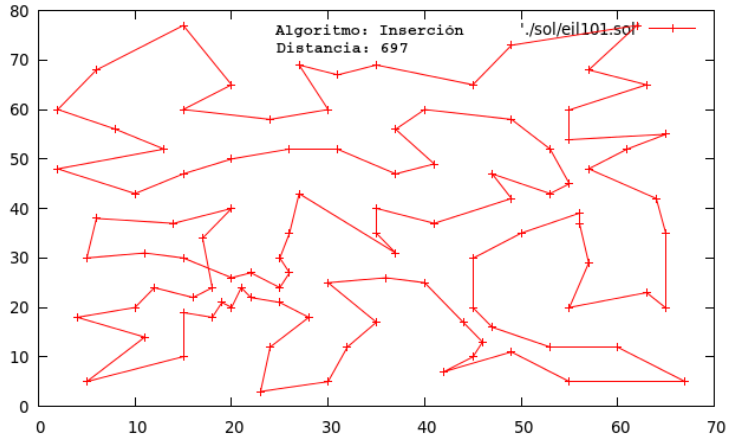
Finalmente, mostraremos los recorridos que se obtienen con las diferentes heurísticas explicadas para algún recorrido con el que testearemos el problema del viajante del comercio. En muchos casos apreciaremos a simple vista qué caminos son mejores, lo que nos dará una idea de la eficacia de los distintos algoritmos:

4.5.1 Recorrido obtenido por el Vecino Más Cercano



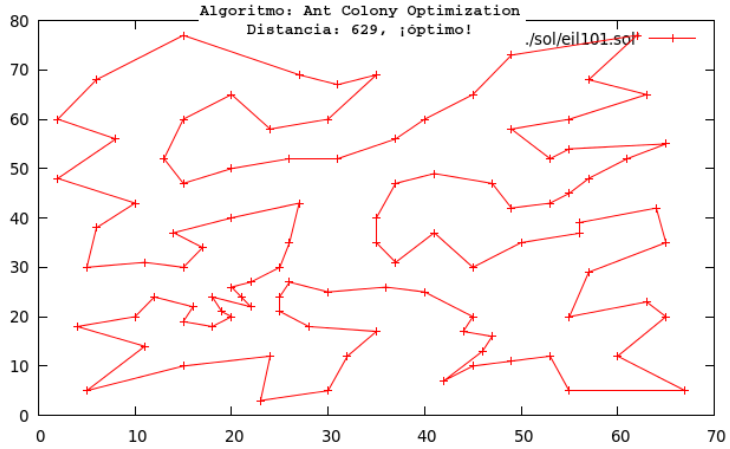
Cabe destacar sobre este algoritmo que el recorrido obtenido es un recorrido con sentido, es decir, no se trata de ningún camino inabordable respecto al coste, y se aprecia a simple vista su buena disposición. También podemos observar el principal defecto de este algoritmo, y es que, tras ir escogiendo siempre la ciudad más cercana a la última visitada, llegamos a situaciones en las que quedan las ciudades muy distanciadas y no queda más remedio que elegir añadir un arco al camino que no nos es favorable. De ahí los arcos de la imagen que resultan intuitivamente demasiado largos, lo que da a entender que sus respectivas ciudades podrían haberse enlazado de otra forma más eficaz a la hora de obtener la solución.

4.5.2 Recorrido obtenido por la Mejor Inserción



Intuitivamente, en este recorrido se puede apreciar una mejora considerable respecto al anterior. Al ir buscando la mejor inserción nos quitamos el problema que teníamos en el algoritmo anterior: ya no vemos esos arcos de mal gusto que intervenían en la imagen previa. Un inconveniente aquí es el de que el recorrido parcial que se va obteniendo es invariante salvo la inserción que se haga con cada ciudad, es decir, la secuencia de ciudades, aunque no es inmutable, marca la tendencia del algoritmo limitando su mejoría.

4.5.3 Recorido obtenido por la Optimización de las Hormigas



Aquí la imagen lo dice todo. Pocas palabras se pueden añadir. Una mera observación basta para ver que el recorrido obtenido es extraordinariamente bueno. De hecho, las hormigas han logrado en este caso la solución óptima para este problema concreto. La imagen corrobora su eficacia.