Algorítmica - Practica 3: Algoritmos Voraces

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

03 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) optentiendo 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.

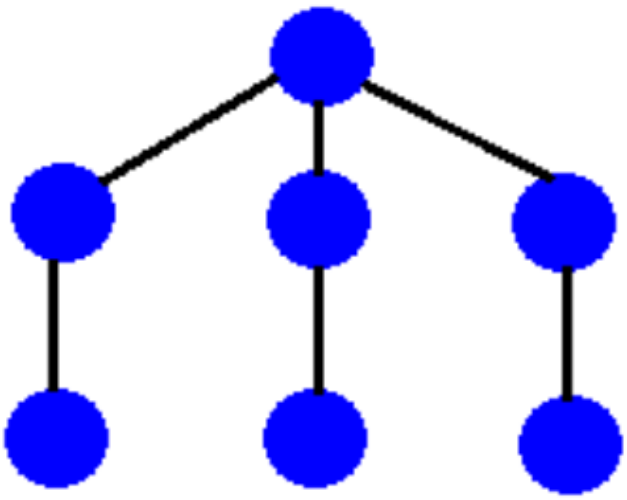

Imagen 1. Árbol sobre el que aplicaremos los algoritmos.

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

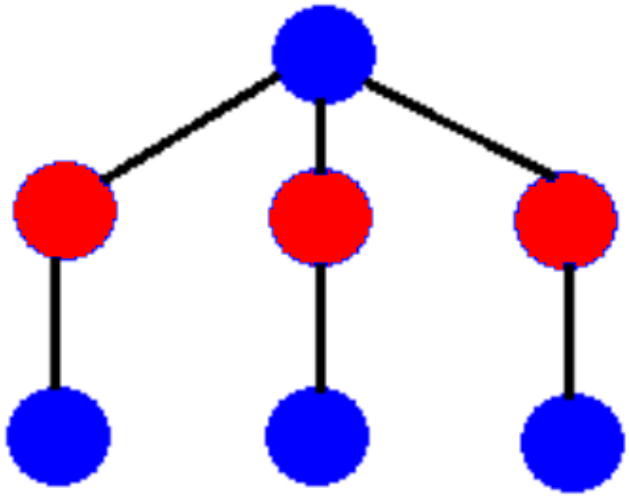

Imagen 2. Recubrimiento minimal para el árbol de la Imagen 1.

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.

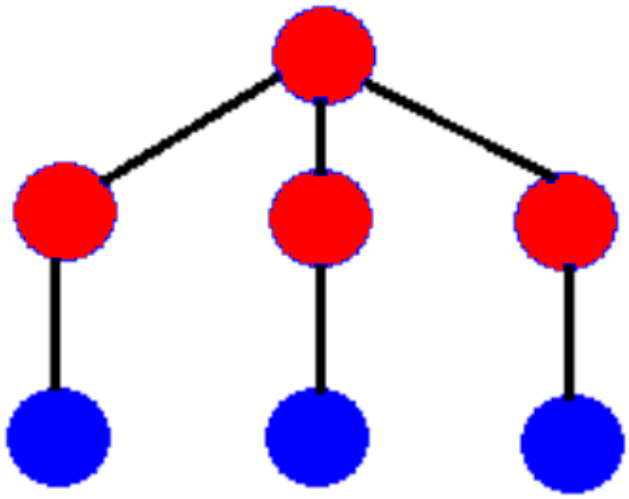

Imagen 3. Un recubrimiento no minimal del árbol de la Imagen 1.

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 U:
        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.
U = []
while not T.isEmpty:
    hojas = T.hojas()
    for hoja in hojas:
        U.append(hoja.parent)
        T.delete([hoja, parent])
```

Su funcionamiento es el siguiente, en cada iteración se calculan las hojas del árbol 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 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 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_1, 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 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, 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 la altura 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.

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.

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':

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

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.