Capítulo 6

Algoritmos voraces.

6.1. Competencias

Las competencias a desarrollar en el tema son:

- **CTEC3** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

6.2. Introducción

Este método también se conoce por *Método devorador* o *Método greedy*. En él sigue presente la idea de división, al igual que en el *Divide y Vencerás*, aunque ahora se refiere a división de la solución y no al problema en sí.

El método es aplicable a problemas cuya solución se pueda obtener a trozos, y la forma de obtener cada trozo se basa en buscar aquel de los posibles trozos, aún no utilizados, que optimiza una función objetivo. Por esta razón este método se aplica en muchos problemas de optimización. Hay que tener en cuenta que la obtención de este trozo de solución, aparentemente óptimo en ese instante, nunca considera lo que pueda ocurrir más adelante y, por tanto, ya se considera definitivo.

De lo naterior, se deduce que los algoritmos voraces se basan en la información que se posee de modo inmediato, sin tener en cuenta las consecuencias que pueden tener en el futuro las decisiones que se toman en un momento puntual. Por esta razón, son fáciles de diseñar y cuando funcionan son bastante eficientes. Por otra parte, hay que resaltar que no todos los problemas cuya solución se basa en este método, se pueden resolver de forma óptima.

Un ejemplo típico es el problema del cambio. Supongamos que tenemos disponibles las siguientes monedas en un sistema monetario: 1 euro, 50 céntimos, 20

céntimos, 10 céntimos, 5 céntimos, 2 céntimos y 1 céntimo y pretendemos diseñar un algoritmo para pagar una cantidad a un cliente, de forma tal que se utilice el menor número de monedas. Si por ejemplo tuviésemos que pagarle 3 euros y 88 céntimos, es muy probable que se utilizasen 3 monedas de euro, una de 50 céntimos, una de 20 céntimos, una de 10 céntimos, una de 5 céntimos, una de 2 céntimos, y una de 1 céntimo. En total serían 9 monedas. Este problema se ha resuelto usando un algoritmo voraz de forma inconsciente. Para ello vamos seleccionando las monedas del mayor valor posible, sin pasarnos de la cantidad establecida, hasta conseguir dicha cantidad. Un algoritmo que resuelva el problema podría ser el siguiente:

Algoritmo $cambio(n)$
  inicio
    $C = \{100, 50, 20, 10, 5, 2, 1\}$
    $S \leftarrow \phi \;\; Solucion$
    $s \leftarrow 0 \;\; suma \;\; parcial$
    mientras $s \neq n$ **hacer**
      $x \leftarrow maximo(C) \;\; tal \;\; que \;\; s + x \leq n$
      si $existe(x)$ **entonces**
        $S \leftarrow S \cup \{x\}$
        $s \leftarrow s + x$
      sino
        devolver $no \;\; encuentro \;\; solucion$
      finsi
    finmientras
  fin

Este algoritmo es voraz porque en cada paso selecciona la mayor de las monedas que se pueda escoger, sin preocuparse si esta decisión es la correcta a la larga. Además no hay vuelta atrás ya que una vez escogida una moneda, ésta forma parte de la solución final.

Este algoritmo es óptimo para los valores dados de las monedas, pero si cambiamos estos valores o si tenemos alguna limitación en la cantidad de monedas disponibles, el algoritmo puede producir una solución que no es óptima. Por ejemplo, si existiesen monedas de 4 céntimos y queremos obtener una cantidad de 8 céntimos, el algoritmo nos daría una solución compuesta por una moneda de 5 céntimos, una de 2 céntimos y otra de un céntimo, cuando la solución optima serían 2 monedas de 4 céntimos. Esta es la razón por la cual no existen monedas de 4 céntimos ni monedas correspondientes a otras cantidades.

Nuestro sistema monetario se dice que es perfecto, ya que para conseguir un cambio determinado, la estrategia de este algoritmo voraz es la que proporciona el menor número de monedas (solución óptima). La mayoría de los sistemas monetarios en el mundo son perfectos.

6.3. El método general.

Los algoritmos voraces se caracterizan por una serie de propiedades que se van a enumerar a continuació. Para una mejor comprensión, dichas propiedades se van a particularizar en el problema del cambio. Las propiedades se pueden resumir en:

- Problema de optimización en el que la solución se construye partiendo de un conjunto de candidatos (monedas disponibles sabiendo que no hay límite en ninguna de ellas).
- A medida que avanzamos en la solución se van acumulando dos conjuntos. Uno contiene candidatos evaluados y seleccionados (monedas ya seleccionadas) y el otro contiene los evaluados y rechazados (monedas con valor superior al cambio que nos queda por obtener y que ya no pueden formar parte de la solución).
- Una función comprueba si los candidatos seleccionados hasta el momento constituyen una solución del problema, ignorando si es óptima por el momento. En el caso del cambio, viendo si las monedas seleccionadas suman la cantidad que hay que pagar.
- Otra función comprueba si un cierto conjunto de candidatos puede hacer crecer el conjunto, añadiendo mas candidatos que obtengan al menos una solución del problema (monedas de menor valor que el cambio que nos queda por obtener). Aquí sigue sin preocuparnos si es óptima o no.
- Función de selección que indica cual es el mejor de los candidatos restantes, que aún no han sido ni aceptados ni rechazados (de todas las monedas que aún se pueden seleccionar, se elige la de mayor valor).
- Función objetivo que proporciona el valor de la solución encontrada (número de monedas empleadas en el cambio).

Para resolver el problema se buscan un conjunto de candidatos que formen una solución y que optimice a la función objetivo. Inicialmente este conjunto está vacío y en sucesivas etapas se añade al conjunto el mejor candidato por medio de la función de selección. Si el conjunto resultante, después de añadir el candidato, no fuese viable, se rechaza dicho candidato. Por el contrario, si el conjunto fuese viable, se añade el candidato al conjunto actual de candidatos seleccionados, y ya formará parte definitiva del mismo. Cada vez que se añade un candidato definitivamente, se comprueba si el conjunto obtenido es una solución del problema. Si el algoritmo voraz proporcionase una solución óptima, cosa que no siempre ocurre, la primera solución obtenida es siempre óptima.

```
Algoritmo voraz(C)
  inicio
    S ← φ  Solucion
    mientras C ≠ φ  y no solucion(S)  hacer
      x ← seleccionar(C)
      C ← C − {x}
      si viable(S ∪ {x})  entonces
        S ← S ∪ {x}
      finsi
    finmientras
    si solucion(S)  entonces
      devolver S
    sino
      devolver No hay solucion
    finsi
  fin
```

El nombre voraz proviene del hecho de seleccionar el *bocado más apetecible* en cada etapa, sin preocuparse por lo que pueda ocurrir más adelante. La función de selección suele estar relacionada con la función objetivo, ya que si queremos optimizar una función objetivo lo lógico es que el candidato seleccionado sea también el que haga crecer más (maximización) o menos (minimización) la función objetivo. Puede ocurrir también que existan varias funciones de selección viables, así que habrá que seleccionar la más adecuada.

En cuanto al tiempo de ejecución de los algoritmos basados en este método, y teniendo en cuenta que:

- El conjunto de candidatos en cada etapa es de cardinal menor al del conjunto de candidatos de partida.
- La función objetivo y la función de selección pueden consumir un tiempo constante o a lo sumo lineal.
- El número de candidatos que formarán parte de la solución será una fracción del conjunto de candidatos de partida.

se puede concluir que dichos algoritmos son de complejidad $O(n^2)$ cuando el tiempo consumido por las funciones objetivo y de selección sea constante, o a lo sumo $O(n^3)$, cuando el tiempo consumido por la funciones objetivo y/o de selección sea lineal.

6.4. Ejemplos.

6.4.1. El problema de la mochila.

Este problema tiene varias versiones. En este apartado se va a analizar la versión más simple, ya que en ella no se imponen restricciones a la hora de obtener la solución. Se dispone de una mochila con un cierto volumen de capacidad, y se tienen una serie de materiales divisibles, ya que se puede usar una fracción de un material (en versiones más complejas del problema, los materiales no son divisibles) que ocupan un volumen y cada uno con un precio por unidad de volumen. La cuestión es llenar la mochila con dichos materiales, de forma tal que se maximice el valor de la misma considerando el coste de los materiales que introduciremos en ella.

Para resolver el problema usando un algoritmo voraz se define el conjunto de los datos del problema y los candidatos que asociaremos a dicho conjunto de datos para obtener la solución. Los datos son:

- El volumen de la mochila V.
- Los n materiales $m_1, m_2, ..., m_n$
- Sus volúmenes $v_1, v_2, ..., v_n$
- Sus precios por unidad de volumen $p_1, p_2, ..., p_n$.

La solución será una tupla de reales $(x_1, x_2, ..., x_n)$, que indicará la cantidad de cada material que introducimos en la mochila, de forma tal que $0 \leq x_i \leq v_i$ y $\sum_{i=1}^{n} x_i = V$. Los candidatos serían pares de la forma (i, x_i) con $1 \leq i \leq n$ y $0 \leq x_i \leq v_i$, que indican la cantidad x_i que se utiliza del material m_i.

En este caso, la solución se podría considerar prácticamente como un subconjunto del conjunto inicial de candidatos del problema, siendo realmente un conjunto de objetos (i, x_i) con $1 \leq i \leq n$. La clave está en decidir qué componentes x_i son nulas, ya que las no nulas, casi siempre tenderán a ser el total v_i disponible de cada material, excepto cuando se use una fracción de un material para completar la mochila. En cada etapa se elige un x_i todavía no seleccionado, y una vez elegido se comprueba si se puede usar en su totalidad (no se rebasa el volumen de la mochila) o se selecciona parcialmente, en cuyo caso sería el último material seleccionado. Una vez seleccionado el material m_i, su volumen habrá que restarselo al volumen disponible en la mochila.

Si la suma de los volúmenes de todos los materiales es igual o inferior al volumen de la mochila ($\sum_{i=1}^{n} v_i \leq V$) está claro que el óptimo consiste en introducir todos los materiales en la mochila.

Los casos interesantes serán aquellos en los que no todos los materiales formen parte de la solución. También está claro que una solución óptima debe llenar por

completo la mochila, ya que en caso contrario siempre se podría añadir algún trozo de material más, incrementando el valor de la mochila. La estrategia general sería seleccionar en cada paso un material, siguiendo un orden adecuado, poniendo la mayor cantidad posible de dicho material en la mochila y acabar cuando ésta esté llena.

Por último queda decidir cual es el criterio de elección del material en cada etapa. Evidentemente, como se quiere maximizar el valor de la mochila, en cada etapa se seleccionará aquel material, aún no seleccionado, que haga crecer el coste de la mochila en la mayor medida posible. Ello implicará que siempre se seleccione el material de mayor precio por unidad de volumen, y en su totalidad si es posible. Como se ha citado antes, la solución puede ser un subconjunto del conjunto de datos del problema. Esto no ocurrirá cuando el último material se seleccione parcialmente para completar la mochila.

Una forma de implementar el algoritmo consistiría en obtener los precios en orden decreciente, ir añadiendo materiales a la mochila en ese orden hasta que ésta esté llena. Para obtener el precio en orden decreciente tenemos dos opciones:

- Obtener los precios en orden decreciente 1 a 1, sin tener que ordenarlos todos.
- Ordenarlos todos.

La primera forma tendría un tiempo de ejecución de orden kn, siendo k el número de materiales elegidos, y el segundo método emplearía un tiempo de orden $nlogn$ si empleásemos un método de ordenación sofisticado. A continuación se detalla el algoritmo. Para implementarlo se usará un vector de estructuras D para guardar las características de cada material, donde:

- $volumen$. Es el volumen disponible de cada material.
- $precio$. Es el precio unitario (por unidad de volumen) de cada material.
- $usado$. Cadena que indica si el material ha sido usado en su totalidad, parcialmente o no se ha usado nada.

```
Algoritmo Mochila(n, V; D; )
  inicio
    resto ← V
    En principio se marcan los materiales como no usados
    para i de 1 a n hacer
      D(i).usado ← "nada"
    finpara
    repetir
      precioMaximo ← 0
      materialMaximo ← 0
```

```
    materialDisponible ← falso
    Se selecciona el material de maximo coste
    para i de 1 a n hacer
      si D(i).usado = "nada" entonces
        materialDisponible ← cierto
        si D(i).precio > precioMaximo entonces
          precioMaximo ← D(i).precio
          materialMaximo ← i
        finsi
      finsi
    finpara
    Comprobamos si el material de maximo coste cabe en la mochila
    si materialDisponible = cierto entonces
      si resto ≥ D(materialMaximo).volumen entonces
        D(materialMaximo).usado ← "total"
        resto ← resto − D(materialMaximo).volumen
      sino
        D(materialMaximo).usado ← "parcial"
        resto ← 0
      finsi
    finsi
  hasta que resto = 0 o materialDisponible = falso
fin
```

6.4.2. Minimización del tiempo de espera.

Supongamos que en un determinado servicio (ventanilla, gasolinera, consulta médica, etc.) se han de atender a n clientes, y de antemano se conoce el tiempo t_i que se va a emplear en atender a cada cliente, y que todos los clientes están en el servicio cuando se comienza a atender al primero. El problema consiste en estimar en qué orden deben ser atendidos los clientes, para que la suma de los tiempos de espera y de atención de todos sea mínima. Evidentemente, si solamente consideramos los tiempos de atención, la suma será siempre la misma, independientemente del orden en el que se atiendan. Por ejemplo, a un médico que realizase un chequeo a los trabajadores de una empresa, le daría igual atender a los clientes en cualquier orden, sin embargo a la empresa no le daría lo mismo, ya que dependiendo del orden, se perderían más o menos horas de trabajo.

Sea t_i el tiempo que tarda en atenderse al cliente atendido en i-ésimo lugar, en ese caso el tiempo T_i que ese cliente está en el servicio sería su tiempo de atención t_i, mas los tiempos de atención de todos los clientes que son atendidos delante de

él:

$$T_i = \sum_{j=1}^{i} t_j$$

Supongamos que se tienen tres clientes, y los tiempos de atención son $t_1 = 5, t_2 = 10, t_3 = 3$ tendríamos 6 posibilidades a la hora de atenderlos. Esas posibilidades serían:

- 1, 2, 3. El tiempo total en servicio sería: $5 + (5 + 10) + (5 + 10 + 3) = 38$.
- 1, 3, 2. El tiempo total en servicio sería: $5 + (5 + 3) + (5 + 3 + 10) = 31$.
- 2, 1, 3. El tiempo total en servicio sería: $10 + (10 + 5) + (10 + 5 + 3) = 43$.
- 2, 3, 1. El tiempo total en servicio sería: $10 + (10 + 3) + (10 + 3 + 5) = 41$.
- 3, 1, 2. El tiempo total en servicio sería: $3 + (3 + 5) + (3 + 5 + 10) = 29$.
- 3, 2, 1. El tiempo total en servicio sería: $3 + (3 + 10) + (3 + 10 + 5) = 34$.

La secuencia óptima es la $3, 1, 2$ cuyo tiempo total en el servicio es 29.

Supongamos que un algoritmo va construyendo la secuencia óptima paso a paso. Después de haber calculado la secuencia óptima $(i_1, i_2, ..., i_m)$ para los m primeros clientes, supongamos que se añade a la misma el cliente j, con tiempo de atención t_j. El crecimiento del tiempo total en el servicio T será:

$$t_{i1} + t_{i2} + t_{i3} + \cdots + t_{im} + t_j$$

que sería el tiempo correspondiente de espera y atención del cliente j que se ha añadido. Para minimizar este crecimiento, dado que un algoritmo voraz no reconsidera sus decisiones y los tiempos previos seleccionados ya no se pueden cambiar, lo único factible es el minimizar t_j. De aquí se deduce que la estrategia voraz a seguir consiste en atender en cada paso al cliente no atendido con menor tiempo de atención. Esto significa atender a los clientes en orden creciente del tiempo de atención, ya que siempre podrá ser mejorada cualquier secuencia en la cual un cliente con mayor tiempo de atención sea atendido antes que otro con menor tiempo de atención. Sin embargo, esta mejora no es posible si se atienden en orden creciente del tiempo de atención.

Para demostrar que el algoritmo es óptimo, podríamos seguir el siguiente razonamiento:

Sea $P = p_1, p_2, p_3, \ldots, p_n$ una permutación cualquiera de los enteros que van de 1 a n, que representa el tiempo de atención a los n clientes. Sea t_{Pi} el tiempo de atención al cliente $i-$ésimo en esa permutación P. El tiempo total en el servicio para esa permutación será:

$$T_p = t_{P1} + (t_{P1} + t_{P2}) + (t_{P1} + t_{P2} + t_{P3}) + \cdots + (t_{P1} + t_{P2} + t_{P3} + \ldots t_{Pn})$$

$$T_p = nt_{P1} + (n-1)t_{P2} + \ldots 2t_{Pn-1} + t_{Pn} = \sum_{k=1}^{n}(n-k+1)t_{Pk}$$

Si P no tiene a los clientes en orden creciente de tiempo de atención, entonces es posible encontrar dos enteros i, j tales que $i < j$ y $t_{Pi} > t_{Pj}$, en este caso se atiende antes al cliente i que al j aunque su tiempo de atención sea superior. Si se intercambiase la posición de estos dos clientes tendríamos una nueva permutación Q, que es la misma que P después de intercambiar p_i y p_j. El tiempo total en el servicio para la permutación Q será:

$$T_Q = (n-i+1)t_{Pj} + (n-j+1)t_{Pi} + \sum_{k=1,k\neq i,j}^{n}(n-k+1)t_{Pk}$$

Extrayendo los términos i y j del sumatorio, T_P se puede expresar como:

$$T_P = (n-i+1)t_{Pi} + (n-j+1)t_{Pj} + \sum_{k=1,k\neq i,j}^{n}(n-k+1)t_{Pk}$$

Si calculamos la diferencia entre T_P y T_Q obtenemos lo siguiente:

$$T_P - T_Q = (n-i+1)(t_{Pi} - t_{Pj}) + (n-j+1)(t_{Pj} - t_{Pi}) = (j-i)(t_{Pi} - t_{Pj}) > 0$$

Lo cual demuestra que T_Q es menor que T_P y por tanto el tiempo se puede mejorar. De aquí se deduce que al ordenar al menos dos elementos desordenados de una permutación dada. se disminuye el tiempo total de espera en el sistema. Por tanto, si se atienden en orden creciente de tiempos, dicha mejora no es posible.

Un problema similar es el que se planteaba en las antiguas cintas de cassete a la hora de establecer el orden de las canciones. Dado que para acceder a una canción, había que escuchar todas las anteriores, el problema de obtener cual sería el orden de las canciones para minimizar el tiempo de escucha más el de espera para escuchar una determinada canción, es similar al planteado en este apartado.

6.4.3. Planificación de tareas a plazo fijo.

En este problema, se tienen un conjunto de n tareas que se quieren realizar, y cada una de ellas se realiza en una unidad de tiempo. Además en cada instante $t = 1, 2, 3, \ldots, n$ solo se puede ejecutar una tarea. La tarea i, con $1 \leq i \leq n$ genera un beneficio b_i solo si se realiza en un instante igual o anterior a t_i. En el caso de que se realizase después no generaría beneficio.

Vamos a ver un ejemplo para clarificar el problema. Supongamos que para $n = 4$, tenemos 4 tareas, $1 \leq i \leq 4$, donde i representa una tarea i-ésima. Se tienen

secuencia de tareas	beneficio
1	50
2	10
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80
4,3	45

Cuadro 6.1: Tabla de secuencias y beneficios para el ejemplo.

los valores de beneficio $b_i = \{50, 10, 15, 30\}$ y de plazos $p_i = \{2, 1, 2, 1\}$ para $1 \leq i \leq 4$. La tabla 6.1 refleja todas las secuencias posibles a considerar y sus respectivos beneficios:

Como se puede observar, solo se han considerado las tareas que se realizan en plazo, ya que el resto no generan beneficios. Por ejemplo, la secuencia $3, 2$ no se considera porque la tarea 2 sería efectuada en el instante $t = 2$, después del plazo $p_2 = 1$, que es el que tiene establecido. Se puede observar que la secuencia que maximiza el beneficio es la $4, 1$, ya que produce un beneficio de 80.

Un conjunto de tareas es factible si existe al menos una secuencia (que se llamará factible) que permite realizar todas las tareas en sus plazos respectivos. Un algoritmo voraz evidente es aquel que construye la secuencia de tareas una por una, incorporando en cada paso la tarea aún no considerada de mayor beneficio, bajo la condición de que la secuencia construida sea factible.

En el ejemplo anterior se selecciona en primer lugar la tarea 1, después la 4. La secuencia $1, 4$ es factible porque puede realizarse en el orden $4, 1$. Seguidamente se probaría con la $1, 4, 3$, que resulta ser no factible ya que la tarea 3 sería rechazada. Finalmente se prueba con la $1, 4, 2$ que tampoco es factible y se rechaza la tarea 2. La solución óptima en este caso es realizar la secuencia de tareas $1, 4$ en el orden forzoso $4, 1$.

Si tenemos un conjunto T de k tareas, y quisieramos ver si es factible, en principio habría que probar las posibles $k!$ permutaciones de ese conjunto para ver si T es factible. Esta comprobación tendría una complejidad excesiva y no sería posible realizarla para un k relativamente grande. Por suerte, se puede demostrar que comprobando solo una permutación $P = (i_1, i_2, \ldots, i_k)$, en la cual las tareas estuviesen ordenadas por orden no decreciente de sus plazos de ejecución, el conjunto T es factible si y solo si la secuencia mencionada es factible. Para demostrar esta

afirmación seguimos dos pasos:

- Demostrar que si la permutación no decreciente es factible, el conjunto T también lo es. Esta demostración es evidente, ya que si las tareas de dicha permutación se pueden realizar sin violar ningún plazo, entonces T es factible porque está formado por las tareas de esa permutación.

- Demostrar que si T es factible, entonces la secuencia ordenada por orden no decreciente (creciente) también lo es. Si T es factible, supongamos que existe una permutación de T factible donde podría haber dos tareas cuyos plazos no estuviesen en orden creciente. En ese caso, es evidente que si intercambiamos esas tareas, la nueva permutación de T seguiría siendo factible. Si aplicamos este intercambio a todas las tareas que no estén en orden creciente, seguiríamos teniendo permutaciones factibles. Continuando de esta forma, podemos llegar a obtener la permutación ordenada no decrecientemente (crecientemente) sin violar ningún plazo. Esto demuestra que si T es factible, es porque como mínimo su permutación en orden no decreciente (creciente) es factible.

También se puede demostrar que algoritmo voraz que utiliza la permutación ordenada no decrecientemente (crecientemente) proporciona una solución óptima al problema.

Para la implementación del algoritmo vamos a suponer que se numeran las tareas en orden no creciente (decreciente) de los beneficos, $b_1 \geq b_2 \geq b_3 \geq \cdots \geq b_n$, de esta forma los beneficios no influyen en la solución final dada por el algoritmo. p es el vector de los plazos de cada una de las tareas. Para poder implementarlo más eficientemente se va a suponer que $n > 0$ y $p_i > 0$ para $1 \leq i \leq n$. Además se va a usar la posición 0 como centinela de los vectores p y s (vector solución del problema que indicará la secuencia de tareas).

Como están en orden decreciente de beneficios, la tarea primera siempre entrará en la solución. Para seguir introduciendo tareas en la solución, cuando se evalúa la tarea i, el algoritmo comprueba si se puede insertar en el vector solución s en el lugar oportuno sin llevar alguna tarea que ya esté en la solución, más allá de su plazo. En este caso la tarea i se inserta en la solución y en caso contrario se rechaza. Hay que resaltar que las tareas quedan insertadas en orden creciente de plazos. Los valores del beneficio no son necesarios para el algoritmo siempre y cuando las tareas estén numeradas en orden decreciente de los beneficios.

Algoritmo $secuencia(p, n; ; k, s)$
inicio
$p(0) \leftarrow 0$
$s(0) \leftarrow 0$
$k \leftarrow 1$

```
s(1) ← 1 La tarea 1 se selecciona siempre
para i de 2 a n hacer  en orden decreciente de los beneficios
  busca tarea y prueba insertarla sin que las ya seleccionadas
  queden fuera de plazo y salgan de la solucion
  r ← k almacena la última tarea seleccionada
  mientras p(s(r)) > p(i)  y  p(s(r)) ≠ r  hacer
    la primera parte del predicado busca la posición de inserción
    la segunda comprueba si la tarea r, ya colocada, puede ser desplazad
    sin violar plazos
    r ← r − 1
  finmientras
  Encuentra la posición de inserción comparando con las
  ya seleccionadas e inserta ó inserta porque no se cumplirían plazos
  si p(s(r)) ≤ p(i)  y  p(i) > r  entonces
    la primera parte del predicado comprueba que ha encontrado
    la posición de inserción
    la segunda comprueba que la tarea se inserta sin violar su plazo
    se inserta i en la posicion r+1
    para j de k a r + 1 inc −1 hacer
      Desplaza una posición las tareas desplazables
      s(j + 1) ← s(j)
    finpara
    Inserta la tarea nueva en la posición r+1
    s(r + 1) ← i
    Pasa a evaluar la siguiente tarea
    k ← k + 1
  finsi
finpara
fin
```

A continuación, en la tabla 6.2 y la figura 6.1 se detalla un ejemplo práctico de funcionamiento del algoritmo:

i	1	2	3	4	5	6
b_i	20	15	10	7	5	3
p_i	3	1	1	3	1	3

Cuadro 6.2: Tabla de secuencias y beneficios para el ejemplo.

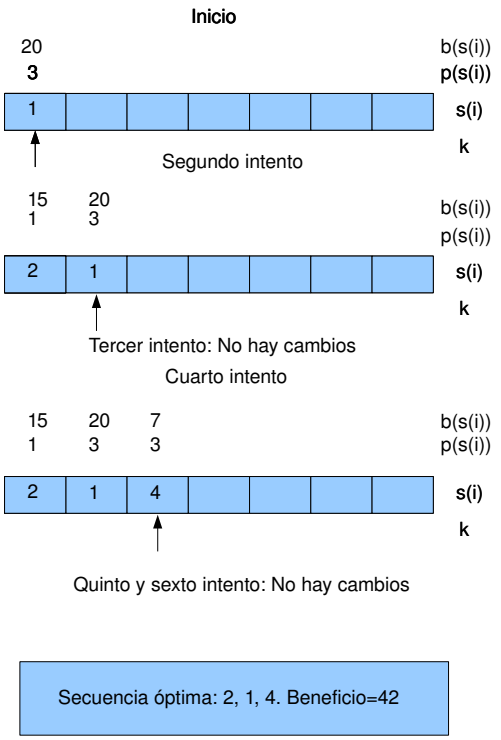

Figura 6.1: Ejemplo de planificación de tareas

6.4.4. Algoritmo de Kruskal.

Este algoritmo ya se estudió en el módulo de grafos, correspondiente a la asignatura de Estructuras de Datos, y en este apartado se va a justificar que se trata de un algoritmo voraz. Se utiliza para estimar el árbol abarcador de coste mínimo en un grafo conexo no dirigido.

Sea L el conjunto de lados, incluidos con los nodos que enlazan, que conforman la solución. Este conjunto es un subconjunto del conjunto de datos del problema (lados y nodos del grafo). Inicialmente el conjunto L no posee ningun lado y cada nodo aislado se considera una componente conexa. A medida que el algoritmo avanza, se van incorporando nuevos lados al conjunto L. El lado que se incorpora en cada etapa es el lado de coste mínimo, aún no seleccionado, que enlaza dos componentes conexas distintas, con lo cual el número de componentes conexas se reduce en una unidad. Por tanto, en una etapa intermedia habrá varias componentes

conexas, y los lados de cada componente conexa forman un árbol abarcador de coste mínimo para los nodos de dicha componente. Cuando finaliza el algoritmo, en L hay una sola componente conexa de coste mínimo que enlaza los nodos del grafo, formando por tanto el árbol de coste mínimo que enlaza los nodos del grafo.

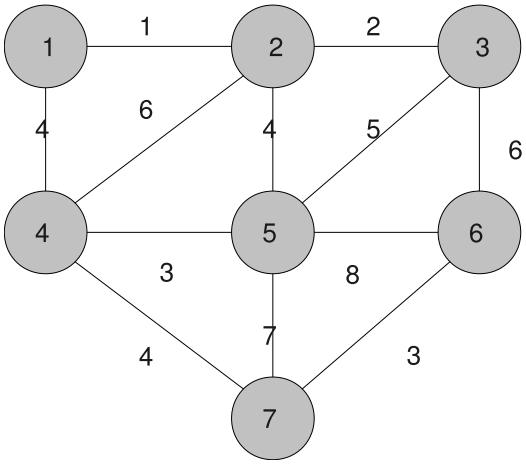

Figura 6.2: Grafo ejemplo para el algoritmo de Kruskal.

Como se ha comentado anteriormente, en la obtención de componentes conexas cada vez más grandes, se van tomando los lados del grafo en orden creciente de pesos, siempre y cuando enlacen dos componentes conexas distintas. Este es un ejemplo donde la elección de un objeto en una etapa, está sometida a la restricción que le imponen los objetos seleccionados en etapas anteriores ya que hay que seleccionar el lado de menor coste con la restricción de que enlace dos componentes conexas distintas. Para ver el funcionamiento del algoritmo, se va a obtener la solución para el grafo de la figura.

Las etapas serían:

- etapa 1. Lado (1, 2). Componentes: (1, 2), (3), (4), (5), (6), (7).
- etapa 2. Lado (2, 3). Componentes: (1, 2, 3), (4), (5), (6), (7).
- etapa 3. Lado (4, 5). Componentes: (1, 2, 3), (4, 5), (6), (7).
- etapa 4. Lado (6, 7). Componentes: (1, 2, 3), (4, 5), (6, 7).
- etapa 5. Lado (1, 4). Componentes: (1, 2, 3, 4, 5), (6, 7).
- etapa 6. Lado (2, 5). Lado rechazado.
- etapa 7. Lado (4, 7). Componentes: (1, 2, 3, 4, 5, 6, 7)

En la implementación del algoritmo, se mantienen varios conjuntos que almacenan los nodos de cada componente conexa. La operación $buscar$ devolverá en qué conjunto (componente conexa) se encuentra un determinado nodo y la operación $fusionar$ unirá dos conjuntos (componentes conexas). El algoritmo quedaría:

Algoritmo $Kruskal(GRAFOG;;GRAFOL)$
inicio
$ordenar(CL)$ `ordena crecientemente el conjunto de lados`
$L \leftarrow \phi$ `Inicialmente ningún lado forma parte de la solucion`
$inicializar\ n\ conjuntos$ `Inicialmente hay tantos conjuntos como nodos`
repetir
$(u, v) \leftarrow$ `Lado mas corto no considerado`
$uconjunto \leftarrow buscar(u)$ `Conjunto al que pertenece nodo u`
$vconjunto \leftarrow buscar(v)$ `Conjunto al que pertenece nodo v`
si $uconjunto \neq vconjunto$ **entonces**
$fusionar(uconjunto, vconjunto)$ `Se fusionan los conjuntos de u y v`
$L \leftarrow L + (u, v)$ `El lado (u,v) se añade al grafo solucion`
finsi
hasta que $L\ tenga\ n-1\ lados$.
fin

6.4.5. El problema del viajante de comercio.

Debido a la sencillez del método devorador, éste puede usarse en algunos problemas para obtener soluciones no-óptimas, pero cercanas al óptimo, en determinadas situaciones. Este es el caso del problema del viajante de comercio. Además este caso servirá para verificar que los algoritmos voraces no siempre conducen a soluciones óptimas en ciertos problemas. Supongamos que se conocen las distancias entre un conjunto de ciudades, y que un viajante debe partir de una de ellas y recorrer todas las demás, volviendo finalmente a la de partida. El problema consiste en ver en qué orden ha de visitar esas ciudades para que la distancia total recorrida sea mínima. Si quisieramos evaluar todas las posibilidades ($n!$) y estimar la mínima, el algoritmo sería de orden $n!$ y sería inviable para un número elevado de ciudades.

Si usamos un grafo para representar las distancias y las ciudades, el problema podría ser abordado por la siguiente idea voraz. En cada iteración seleccionamos el lado más corto aún no seleccionado que cumpla las dos condiciones siguientes:

- No formar un ciclo con los lados ya seleccionados, excepto en la última iteración para cerrar el recorrido.

- No es el tercer lado que incide en el mismo nodo.

$$\begin{pmatrix} 0 & 3 & 10 & 11 & 7 & 25 \\ 3 & 0 & 6 & 12 & 8 & 26 \\ 10 & 6 & 0 & 9 & 4 & 20 \\ 11 & 12 & 9 & 0 & 5 & 15 \\ 7 & 8 & 4 & 5 & 0 & 18 \\ 25 & 26 & 20 & 15 & 18 & 0 \end{pmatrix}$$

Si consideramos el grafo dado por la matriz de conexión reflejada, los lados se seleccionarían en el siguiente orden: (1,2), (3,5), (4,5), (2,3), (4,6), (1,6) para formar el ciclo (1, 2, 3, 5, 4, 6, 1) de longitud 58. El lado (1,5) sería rechazado al formar un ciclo incompleto (1, 2, 3, 5, 1) y además por ser el tercer lado que incide en el nodo 5. Se puede apreciar que el ciclo óptimo es el (1, 2, 3, 6, 4, 5, 1) que tiene una longitud de 56.

Finalmente, hay que resaltar dicha estrategia puede que no obtenga solución al problema. Eso podría ocurrir para un grafo no denso, como es el caso de una red de carreteras. Ello se debe a que se puede llegar a un callejón sin salida en el cual, en una determinada etapa intermedia, no encontramos ningún lado que no forme ciclo, o que los lados que no forman ciclos, son los terceros lados incidentes en un nodo.

6.5. Cuestiones sobre el tema

1. ¿De dónde procede el nombre de voraz en los algoritmos voraces?.

2. Explica en qué consiste un algoritmo voraz, detallando los distintos pasos que sigue un algoritmo voraz.

3. ¿Cual es el orden de complejidad habitual en los algoritmos voraces?. Justifica tu respuesta. (No basta con decir el orden)

4. Los algoritmos voraces son $O(n^2)$ o $O(n^3)$. De qué depende de que sean de un orden o de otro.

5. ¿Se obtiene con un algoritmo voraz una solución óptima en todos los casos?. Justifica tu respuesta e indica un caso donde se verifique tu respuesta.

6. ¿Porqué el algoritmo tradicional del cambio es un algoritmo voraz?.

7. ¿Qué es un sistema monetario perfecto?.

8. ¿Cómo se obtiene la solución al problema de la mochila?

9. ¿Qué condición se ha de cumplir para que el algoritmo voraz de la mochila proporcione una solución óptima?.

10. ¿En qué casos sería la solución al problema de la mochila un subconjunto de los datos de partida?. Justifica tu respuesta.

11. ¿En qué casos sería la solución al problema de la mochila no sería un subconjunto de los datos de partida?. Justifica tu respuesta.

12. Si en una consulta médica se tuviesen que atender a una serie de personas y se conoce el tiempo de atención que necesita cada una. ¿En qué orden deberían ser atendidos para minimizar el tiempo en el que el médico está en la consulta?

13. Si tuvieses que almacenar en una cinta de casette una serie de canciones de distinta duración, ¿En qué orden las almacenarías?. Justifica tu respuesta. Recuerda que para escuchar una canción se han de recorrer todas las anteriores.

14. Describe cómo se obtendría la solución óptima en el problema de la planificación de tareas a plazo fijo.

15. En el algoritmo voraz de la planificación de tareas a plazo fijo, ¿cual es la condición necesaria y suficiente para que un conjunto de tareas sea factible?

16. ¿Cómo se demuestra que si un conjunto de tareas es factible, la permutación en orden creciente de plazos de dicho conjunto también lo es?

17. En el algoritmo voraz de la planificación de tareas a plazo fijo ¿cuál es la única tarea que siempre formaría parte de la solución?. Justifica tu respuesta.

18. Describe como funciona el algoritmo de Kruskal indicando porqué es un algoritmo voraz.

19. ¿Podría obtenerse más de una solución en el algoritmo de Kruskal?. Justifica tu respuesta.

20. ¿Porqué no podría haber más de n-1 lados en la solución del algoritmo de Kruskal?

21. Explica en qué consiste el algoritmo voraz para resolver el problema del viajante de comercio.

22. ¿Es óptima la solución obtenida por el algoritmo voraz del viajante de comercio?.

23. ¿En qúe tipo de grafos es posible que el algoritmo voraz del viajante de comercio no proporcione una solución?