Metodología y Tecnología de la Programación

Ingeniería en Informática

Curso 2008-2009

Esquemas algorítmicos. Algoritmos voraces

Yolanda García Ruiz **D228** **ygarciar@fdi.ucm.es**
Jesús Correas **D228** **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [NN98], [BB97] y notas de S. Estévez y R. González del Campo)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [NN98]: capítulo 4
  - [BB97]: capítulo 6
- Bibliografía complementaria:
  - [GV00]: capítulo 4
- Ejercicios resueltos:
  - [MOV04]: capítulo 12

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Algoritmos voraces

- Se basan en tomar decisiones basadas en la información disponible en cada momento
- No tienen en cuenta los efectos de las decisiones en el futuro
- Nunca se reconsidera una decisión tomada
- Suelen utilizarse para resolver problemas de optimización
- es importante demostrar que la solución que propone el algoritmo es la óptima
- Son fáciles de implementar y eficientes
- Pero hay muchos problemas que no se pueden resolver correctamente con este enfoque

Un ejemplo: Devolución del cambio

- El problema consiste en diseñar un algoritmo que devuelva una cantidad a un cliente utilizando el mínimo número de monedas
- Se dispone de suficientes monedas de todos los tipos
- Hay monedas de 1, 5, 10, 20 y 50 céntimos y 1 y 2 euros.
- Por ejemplo, si tenemos que devolver 3,93 euros, el número mínimo de monedas es 7:
  - 1 moneda de 2 euros
  - 1 moneda de 1 euro
  - 1 moneda de 50 céntimos
  - 2 monedas de 20 céntimos
  - 1 moneda de 2 céntimos
  - 1 moneda de 1 céntimo
- ¿Cómo funciona el algoritmo que utilizamos a diario?

Un ejemplo: Devolución del cambio (cont.)

fun devolver_cambio(n) // n es el importe a devolver (con dos dec.)
  C $\leftarrow$ {2,1,0.5,0.2,0.05,0.02,0.01}
  S $\leftarrow \varnothing$
  s $\leftarrow$ 0
  mientras s $\neq$ n **hacer**
    x $\leftarrow \langle$ el mayor elemento de C tal que $s + x \leq n \rangle$
    si no existe el elemento x **entonces**
      devolver solución no encontrada
    fin si
    S $\leftarrow$ S $\cup$ {una moneda de valor x}
    s $\leftarrow$ s+x
  fin mientras
  devolver S
fin fun

Características generales de los algoritmos voraces

- Los algoritmos voraces se caracterizan por las siguientes propiedades:
  - El objetivo es resolver un problema de forma óptima. Para construir la solución del problema, se dispone de un **conjunto de candidatos**
  - Durante la evolución del algoritmo, se mantienen dos conjuntos:
    - ★ Candidatos ya considerados y seleccionados para la solución
    - ★ Candidatos ya considerados y rechazados
  - Existe una función que determina si un conjunto de candidatos es **solución** del problema
  - Existe una función de **factibilidad** que determina si un conjunto de candidatos es factible (si añadiendo más candidatos es posible construir una solución)
  - Existe una función de **selección** que indica qué candidato de los restantes es el más prometedor
  - Existe una función **objetivo** que proporciona el coste de una solución. Esta función es la que el algoritmo pretende optimizar (maximizar o minimizar). Esta función no aparece explícitamente en el algoritmo

Características generales de los algoritmos voraces (cont.)

- Inicialmente los conjuntos de elementos seleccionados y rechazados están vacíos.
- En cada paso del algoritmo:
  1. se considera el mejor candidato sin considerar a los restantes, exclusivamente mediante la función de selección
  2. Si el conjunto de candidatos seleccionados extendido con este nuevo candidato no es factible, se añade al conjunto de candidatos rechazados
  3. Si es factible, se añade al conjunto de candidatos seleccionados
  4. Se comprueba si este conjunto es solución del problema. Si no lo es, se realiza un nuevo paso del algoritmo (se vuelve a 1)
- Si el algoritmo voraz funciona correctamente, la primera solución que encuentre de esta manera es una solución óptima
- Los algoritmos voraces nunca reconsideran las decisiones tomadas

Un ejemplo: Devolución del cambio (cont.)

- En el caso del cambio de monedas, las características particulares son las siguientes:
- Los candidatos son las monedas de distinto valor, en nuestro caso en cantidad ilimitada (aunque el conjunto debe ser finito)
- La función solución comprueba si el valor de las monedas es **exactamente** el importe a pagar
- La función de factibilidad comprueba si el valor total de las monedas seleccionadas **no sobrepasa** el importe a pagar
- La función de selección toma la moneda **de valor más alto que quede** en el conjunto de candidatos
- La función objetivo es el **número de monedas** utilizadas en la solución. En este caso el problema de optimización que se resuelve debe *minimizar* la función objetivo

Esquema general de los algoritmos voraces

fun voraz(C) // C es el conjunto de candidatos
  S $\leftarrow \varnothing$
  mientras $C \neq \varnothing$ **Y** $\neg solucion(S)$ **hacer**
    x $\leftarrow$ $seleccionar(C)$
    C $\leftarrow$ $C \setminus \{x\}$
    si $factible(S \cup \{x\})$ **entonces**
      S $\leftarrow$ $S \cup \{x\}$
    fin si
  fin mientras
  si $solucion(S)$ **entonces**
    devolver S
  si no
    devolver "no hay soluciones"
  fin si
fin fun

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Árboles de recubrimiento mínimo

- Sea $G = (N, A)$ un grafo conexo no dirigido, donde N es el conjunto de nodos y A el conjunto de aristas
- Cada arista tiene un *peso* (*coste*, *longitud*, *distancia*), una cantidad no negativa.
- El problema consiste en encontrar el conjunto de aristas que permiten comunicar todos los nodos entre sí y tal que la suma de los pesos de estas aristas es mínimo
- Un **árbol de recubrimiento** (spanning tree) de un grafo G es un subgrafo de G tal que contiene todos los vértices y un subconjunto de las aristas y forma un árbol (grafo conexo acíclico)
- El problema a resolver es por tanto encontrar el **árbol de recubrimiento mínimo**
- Esta representación es la más apropiada para diversos problemas, como por ejemplo obtener el trazado de coste mínimo de una red de comunicaciones (kilómetros de carretera o fibra óptica)

Árboles de recubrimiento mínimo (cont.)

- Las características de este problema que se pueden ajustar al método voraz son:
- Los candidatos son las aristas del grafo G
- La función solución consiste en determinar si el conjunto de aristas seleccionadas es un árbol de recubrimiento de G
- Un conjunto de aristas es factible si no contiene ningún ciclo
- La función de selección depende del algoritmo concreto
- La función objetivo es la suma de los pesos de las aristas seleccionadas. Debe minimizarse la función objetivo

Árboles de recubrimiento mínimo (cont.)

- El esquema general de un algoritmo voraz para obtener el árbol de recubrimiento mínimo de un grafo es el siguiente:

  F $\leftarrow \varnothing$
  mientras F no es solución **hacer**
    $e \leftarrow \langle$seleccionar una arista de acuerdo a alguna consideración de optimalidad local$\rangle$
    si al añadir e a F no se crea un ciclo **entonces**
      $F \leftarrow F \cup \{e\}$
    fin si
    si F es un árbol de recubrimiento **entonces**
      devolver F
    fin si
  fin mientras

- A continuación veremos dos algoritmos específicos

Algoritmo de Prim

- Dado el grafo $G = (N, A)$, el algoritmo comienza con un conjunto de aristas F inicialmente vacío y un conjunto de nodos Y con un nodo arbitrario, $Y = \{n_1\}$
- Definimos como *nodo más cercano a* Y el nodo de $N \setminus Y$ que está conectado a un nodo de Y por la arista de menor longitud
- Se añade a Y el nodo más cercano a Y (lo denominamos n_2), y se añade a F la arista de longitud mínima que une n_2 a un nodo de Y
- Se repite este proceso hasta que $Y = N$

Algoritmo de Prim (cont.)

- Esquema de alto nivel del algoritmo:

fun prim(G) $//G = (N, A)$
  $F \leftarrow \varnothing$
  $Y \leftarrow \{n_1\}$
  mientras F no es solución **hacer**
    $n_2 \leftarrow \langle$seleccionar el nodo de $N \setminus Y$ más cercano a $Y\rangle$
    $Y \leftarrow Y \cup \{n_2\}$
    $F \leftarrow F \cup \{(n_2, n_Y)\}$ $//n_Y$ es el nodo de Y con el que se conecta n_2
    si $Y = N$ **entonces**
      devolver F
    fin si
  fin mientras
fin fun

- Las funciones de selección y factibilidad se hacen en un solo paso: al tomar un nodo de $N \setminus Y$ se garantiza que no se crea un ciclo

Algoritmo de Prim (cont.)

- Para definir el algoritmo detallado, debemos utilizar una representación adecuada de los datos para las tareas que se han dejado indicadas en el algoritmo anterior
- Vamos a representar el grafo mediante su matriz de adyacencia:

$$A[i][j] = \begin{cases} \text{distancia de la arista} & \text{si existe una arista entre } n_i \text{ y } n_j \\ \infty & \text{si no existe arista entre } n_i \text{ y } n_j \\ 0 & \text{si } i = j \end{cases}$$

- Además, mantenemos dos vectores de tamaño n (número de nodos):
  - $mas\_cercano[i]$, el índice del nodo de Y más próximo a n_i
  - $distancia[i]$, distancia de la arista existente entre n_i y el nodo al que se refiere $mas\_cercano[i]$
- El algoritmo detallado es el siguiente:

Algoritmo de Prim (cont.)

```
 1: proc prim(A[1..N,1..N], F)
 2:     crear mas_cercano[2..N], distancia[2..N]
 3:     F ← ∅
 4:     desde j ← 2 hasta N hacer
 5:         mas_cercano[j] ← 1 ; distancia[j] ← A[j,1]
 6:     fin desde
 7:     desde i ← 1 hasta N-1 hacer
 8:         min ← ∞
 9:         desde j ← 2 hasta N hacer
10:             si 0 ≤ distancia[j] < min entonces min ← distancia[j] ; prox ← j
11:         fin desde
12:         F ← F ∪ {(mas_cercano[prox],prox)}
13:         distancia[prox] ← -1
14:         desde j ← 2 hasta N hacer
15:             si A[j,prox] < distancia[j] entonces
16:                 distancia[j] ← A[j,prox] ; mas_cercano[j] ← prox
17:             fin si
18:         fin desde
19:     fin desde
20: fin proc
```

Algoritmo de Prim (cont.)

- Es fácil demostrar que el algoritmo de Prim proporciona un árbol (un grafo conexo sin ciclos), pues siempre se añade una arista con un nodo nuevo, no existente anteriormente en Y
- Como este árbol contiene los mismos nodos que el grafo original G, es un árbol de recubrimiento
- Sin embargo, no es evidente que el árbol de recubrimiento sea mínimo, por lo que tenemos que demostrarlo formalmente
- Sea un grafo no dirigido $G = (N, A)$. Un subconjunto de aristas $F \subset A$ se denomina **prometedor** si se le pueden añadir aristas para formar un árbol de recubrimiento mínimo
- Primero vamos a demostrar lo siguiente:
  Lema: Sea $G = (N, A)$ un grafo conexo no dirigido, sea F un subconjunto prometedor de A, y sea Y el conjunto de nodos conectados por las aristas de F.
  Si a es una arista de peso mínimo que conecta un nodo de Y con un nodo en $N \setminus Y$, entonces $F \cup \{a\}$ también es prometedor

Algoritmo de Prim: Demostración de optimalidad

- Como F es prometedor, tiene que existir un conjunto F' tal que $F \subseteq F'$ y que (N, F') sea un árbol de recubrimiento mínimo
- Si $a \in F'$, entonces $F \cup \{a\} \subseteq F'$ y por tanto $F \cup \{a\}$ es prometedor, con lo que el lema ya estaría demostrado
- Si no, como (N, F') es un árbol de recubrimiento, $F' \cup \{a\}$ debe contener un único ciclo y a debe pertenecer a dicho ciclo.
- Por tanto, debe existir otra arista a' en F' que conecte un nodo de Y con un nodo de $N \setminus Y$
- Si eliminamos la arista a' de $F' \cup \{a\}$ el ciclo desaparece, por lo que $F' \cup \{a\} \setminus \{a'\}$ es un árbol de recubrimiento.
- Como el peso de a es el mínimo de todas las aristas que conectan nodos de Y con nodos de $N \setminus Y$, entonces el peso de a debe ser menor o igual que el peso de a'.

Algoritmo de Prim: Demostración de optimalidad (cont.)

- De hecho, deben ser iguales, pues F' es un árbol de recubrimiento mínimo, y de lo contrario $F' \cup \{a\} \setminus \{a'\}$ tendría menor peso total que F', lo que sería una contradicción
- Por tanto, $F' \cup \{a\} \setminus \{a'\}$ es un árbol de recubrimiento mínimo
- Como $F \cup \{a\} \subseteq F' \cup \{a\} \setminus \{a'\}$, $F \cup \{a\}$ es prometedor, con lo que **queda demostrado este lema**
- A partir de este lema podemos demostrar el siguiente

  Teorema: El algoritmo de Prim siempre produce un árbol de recubrimiento mínimo
- Primero demostramos por inducción que el conjunto F es prometedor en cada iteración del bucle **desde** externo (líneas 7-19)

Algoritmo de Prim: Demostración de optimalidad (cont.)

- **Caso base:** el conjunto vacío (valor inicial de F) siempre es prometedor
- **Hipótesis de inducción:** suponemos que, después de una iteración del bucle, el conjunto de aristas seleccionado hasta el momento F es prometedor
- **Paso de inducción:** Debemos demostrar que $F \cup \{a\}$ es prometedor, donde a es la arista seleccionada en la siguiente iteración. Como esta arista es la de menor peso que conecta un nodo de Y con un nodo en $N \setminus Y$, $F \cup \{a\}$ es prometedor, por el lema anterior. Esto completa la demostración por inducción
- Por tanto, el conjunto final de aristas es prometedor. Como este conjunto es un árbol de recubrimiento de G, **éste es un árbol de recubrimiento mínimo, c.q.d.**
- ¿Cuál es la complejidad de este algoritmo?

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Algoritmo de Kruskal

- En este caso el funcionamiento es algo diferente:
  - Inicialmente se crean subconjuntos disjuntos de N, cada uno con un solo nodo del grafo $G = (N, A)$
  - A continuación se inspeccionan las aristas en orden no decreciente de peso
  - Si una arista conecta dos nodos en conjuntos disjuntos, se añade al conjunto de aristas del árbol, y los conjuntos de ambos nodos se unen
  - Este proceso continúa hasta que todos los nodos están en un solo conjunto
- Como en el caso anterior, para implementar este algoritmo es necesario representar la información mediante las estructuras de datos adecuadas
- El grafo se representa mediante el conjunto de aristas A, formado por tuplas con los índices de los nodos y el peso de la arista. De esta forma, (i, j, p) representa una arista entre n_i y n_j con peso p. Los atributos de estas tuplas los denominamos `origen`, `destino` y `peso`, respectivamente.

Algoritmo de Kruskal (cont.)

- Los nodos están representados por sus índices (desde 1 hasta n)
- Utilizaremos un TAD para representar los conjuntos disjuntos. Cada nodo n_i se representa por su índice i. Las operaciones a realizar sobre el TAD son:
  - `inicializar(n)` para inicializar n conjuntos disjuntos, cada uno de ellos con un índice i, $i = 1...n$
  - `p = buscar(i)` devuelve como resultado el conjunto en el que está contenido el nodo n_i con índice i
  - `unir(p,q)` realiza la unión de los conjuntos disjuntos p y q
  - `iguales(p,q)` devuelve *cierto* si p y q se refieren al mismo conjunto
- El algoritmo es el siguiente:

Algoritmo de Kruskal (cont.)

```
//n es el número de nodos, y A el conjunto de aristas (m aristas)
//F es el conjunto de aristas del árbol resultante
proc kruskal(n,A[1..m], F)
   ordenar(A) // ordena A en orden no decreciente de peso
   F ← ∅
   inicializar(n)
   i ← 0
   mientras |F| < n-1 hacer
      i ← i+1
      p ← buscar(A[i].origen) // conjunto disjunto que contiene al origen
      q ← buscar(A[i].destino) // conjunto disjunto que contiene al destino
      si ¬ iguales(p,q) entonces
         unir(p,q)
         F ← F ∪{(A[i].origen, A[i].destino)}
      fin si
   fin mientras
fin proc
```

Algoritmo de Kruskal (cont.)

- Una posible implementación del TAD para los conjuntos disjuntos es la siguiente
- Se requiere el uso de un *array* global `disjuntos[1..N]`
- Cada conjunto está representado por el índice más pequeño de los nodos del conjunto
- Las operaciones del TAD se pueden implementar como sigue:

```
proc inicializar(N)
  desde j ← 1 hasta N hacer
    disjuntos[j] ← j
  fin desde
fin proc

fun buscar(i)
  j ← i
  mientras disjuntos[j] ≠ j hacer
    j ←disjuntos[j]
  fin mientras
  devolver j
fin fun
```

```
proc unir(p,q)
  si p<q entonces
    disjuntos[q] ← p
  si no
    disjuntos[p] ← q
  fin si
fin proc

fun iguales(p,q)
  devolver p=q
fin fun
```

Algoritmo de Kruskal: Estudio de complejidad

- En el caso peor, la complejidad de las funciones `inicializar(n)` y `buscar(i)` es lineal, mientras que las otras son constantes
- Por tanto, la complejidad del algoritmo de Kruskal en el caso peor es cuadrática, pero **respecto al número de aristas**
- En el caso peor, un grafo de entrada de n nodos puede tener $m = n(n-1)/2$ aristas
- Por consiguiente, si todos los nodos están conectados, la complejidad del algoritmo sería cúbica respecto al número de nodos
- En [NN98] se pueden encontrar otras implementaciones del TAD para los conjuntos disjuntos con una complejidad en $\Theta(m\ lg\ m)$ o incluso mejor, lo que daría lugar a una complejidad en el caso peor del algoritmo de Kruskal en $\Theta(n^2 lg\ n)$ respecto al número de nodos
- Dependiendo de la densidad del grafo, el algoritmo de Kruskal puede ser más eficiente que el de Prim, pues si el número de aristas es próximo al número de nodos, su complejidad estaría más próxima a $\Theta(n\ lg\ n)$

Algoritmo de Kruskal: Demostración de optimalidad

- Como en el caso anterior, debemos demostrar que el algoritmo de Kruskal siempre devuelve el árbol de recubrimiento mínimo
- Primero vamos a demostrar un paso del algoritmo mediante el siguiente

  Lema: Sea $G = (N, A)$ un grafo conexo no dirigido, sea F un subconjunto prometedor de A, y sea a un arco de peso mínimo en $A \setminus F$ tal que $F \cup \{a\}$ no tiene ciclos. Entonces $F \cup \{a\}$ es prometedor
- La demostración es similar a la del lema anterior. Como F es prometedor, existe F' tal que $F \subseteq F'$ y (N, F') es un árbol de recubrimiento mínimo.
- Si $a \in F'$, entonces $F \cup \{a\} \subseteq F'$ es prometedor.
- Si $a \notin F'$, $F' \cup \{a\}$ tiene un único ciclo en el que participa a. Como $F \cup \{a\}$ no tiene ciclos, debe haber otra arista $a' \in F'$ que forme parte del ciclo y $a' \notin F$. Es decir, $a' \in A \setminus F$.
- Como a tiene peso mínimo en $A \setminus F$, el peso de a no es mayor al peso de a'
- Si eliminamos a' de $F' \cup \{a\}$, $F' \cup \{a\} \setminus \{a'\}$ es un árbol de recubrimiento. Además, es mínimo, pues el peso de a no es mayor al peso de a'
- Por tanto, $F \cup \{a\}$ es prometedor, c.q.d.

Algoritmo de Kruskal: Demostración de optimalidad (cont.)

Teorema: El algoritmo de Kruskal siempre produce un árbol de recubrimiento mínimo

- Lo demostramos por inducción.
- **Caso base:** el conjunto vacío (valor inicial de F) siempre es prometedor
- **Hipótesis de inducción:** suponemos que, después de una iteración del bucle, el conjunto de aristas seleccionado hasta el momento F es prometedor
- **Paso de inducción:** Debemos demostrar que $F \cup \{a\}$ es prometedor, donde a es la arista seleccionada en la siguiente iteración. La arista seleccionada une nodos que no están conectados por ninguna arista de F, por lo que no se genera un ciclo. Además, a es la arista de peso mínimo de $A \setminus F$. Por tanto, aplicando el lema anterior, $F \cup \{a\}$ es prometedor
- Por tanto, el conjunto final de aristas es prometedor. Como este conjunto es un árbol de recubrimiento de G, **éste es un árbol de recubrimiento mínimo, c.q.d.**

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Caminos mínimos. Algoritmo de Dijkstra

- El problema consiste en encontrar el camino de coste mínimo desde un nodo determinado a todos los demás nodos en un grafo dirigido $G = (N, A)$
- Suponemos que existe al menos un camino desde el nodo origen a todos los demás
- La solución que se presenta fue propuesta por Dijkstra en 1959
- Este algoritmo es similar al algoritmo de Prim visto anteriormente

Caminos mínimos. Algoritmo de Dijkstra (cont.)

- Primero se inicializa un conjunto de nodos Y que contiene solamente el nodo de origen, $Y = \{n_1\}$, y el conjunto de aristas F inicialmente vacío
- Se elige el nodo n más próximo a n_1 (unido por la arista de menor peso).

  Esta arista es claramente el camino mínimo de n_1 a n. Por tanto, se añade n a Y y (n_1, n) a F
- A continuación se comprueban los caminos desde n_1 hasta nodos de $N \setminus Y$ que tengan como nodos intermedios elementos de Y.

  El camino más corto de éstos es un camino mínimo (esto hay que demostrarlo). Se añade el nodo final de este camino a Y y la última arista del camino a F
- Se repite este proceso hasta que $Y = N$. Cuando ocurre esto, F contiene las aristas de los caminos mínimos desde n_1 a los demás nodos.

Caminos mínimos. Algoritmo de Dijkstra (cont.)

- La estructura de alto nivel del algoritmo es la siguiente:

$Y \leftarrow \{n_1\}$
$F \leftarrow \varnothing$
mientras no se ha resuelto el problema **hacer**
  Seleccionar un nodo n de $N \setminus Y$ tal
  que tenga un camino mínimo desde n_1
  utilizando exclusivamente nodos intermedios de Y

  Añadir n a Y
  Añadir a F la arista que conecta con n

  si Y=N **entonces**
    se ha resuelto el problema
  fin si
fin mientras

Caminos mínimos. Algoritmo de Dijkstra (cont.)

- Para desarrollar en detalle este algoritmo se utilizan las siguientes estructuras de datos:
- $D[i]$ es la longitud total del camino mínimo desde n_1 hasta n_i pasando solamente por nodos de Y.
- El grafo de entrada está representado mediante la matriz de adyacencia, como en el algoritmo de Prim.
- Además, para poder proporcionar el conjunto de aristas con los caminos mínimos obtenidos, es necesario un *array conecta*$[i]$ que contiene el índice del nodo n de Y tal que la arista (n, n_i) es la última arista del camino mínimo de n_1 hasta n_i pasando solamente por nodos en Y
- El algoritmo detallado es el siguiente (en color rojo aparece la parte del algoritmo relacionada con la obtención de las aristas):

Caminos mínimos. Algoritmo de Dijkstra (cont.)

```
proc dijkstra(A[1..n,1..n], D[2..n], F)
   crear conecta[2..n]
   C ← {2, ..., n} ; F ← ∅
   desde i← 2 hasta n hacer
      D[i] ← A[1,i] ; conecta[i] ← 1
   fin desde
   desde k← 1 hasta n-1 hacer
      min ← ∞
      para todo i ∈ C hacer
         si D[i] < min entonces min ← D[i] ; prox ← i
      fin desde
      F ← F ∪{(conecta[prox],prox)}
      C ← C \{prox}
      para todo i ∈ C hacer
         si D[prox]+A[prox,i] < D[i] entonces
            D[i] ← D[prox]+A[prox,i] ; conecta[i] ← prox
         fin si
      fin desde
   fin desde
fin proc
```

Algoritmo de Dijkstra: Demostración de optimalidad

- Sea el camino $\hat{c} = \{n_1, n_2, ..., n_{k-1}, n_k\}$. Se dice que $\hat{c}$ es *especial* si $\{n_1, n_2, ..., n_{k-1}\} \subseteq Y$
- Vamos a demostrar el siguiente

  Teorema: El algoritmo de Dijkstra halla los caminos más cortos desde un nodo origen a todos los demás nodos del grafo
- Demostramos por inducción:
  a) si un nodo $n_i, i \neq 1$, está en Y, $D[i]$ contiene la longitud del camino mínimo de n_1 a n_i
  b) si $n_i, i \neq 1$, no está en Y, $D[i]$ contiene la longitud del camino especial más corto de n_1 a n_i
- **Caso base:** En el caso inicial, $Y = \{n_1\}$, y tanto a) como b) se cumplen de forma trivial
- **Hipótesis de inducción:** suponemos que, antes de una iteración del bucle, D contiene los caminos mínimos desde n_1 a todos los nodos en Y, y los caminos especiales más cortos a los nodos de $N \setminus Y$

Algoritmo de Dijkstra: Demostración de optimalidad (cont.)

- **Paso de inducción: a)**
  - en cada iteración del bucle, se elige el nodo $n_i \in N \setminus Y$ tal que el camino especial de n_1 a n_i es mínimo. Debemos demostrar que este camino es efectivamente el camino mínimo de n_1 a n_i en el grafo G
  - Supongamos que en el camino mínimo de n_1 a n_i hay algún nodo que no pertenece a Y. Sea x el primer nodo que no pertenece a Y en este camino.
  - Como el camino desde n_1 hasta x es un camino especial, $D[x]$ es la distancia de este camino (por la parte b) de la hipótesis de inducción).
  - Como las distancias son no negativas, la distancia total a n_i a través de x no es menor a $D[x]$
  - Por otra parte, $D[x]$ no es menor a $D[n_i]$ pues el algoritmo ha seleccionado a n_i
  - Por tanto, la longitud del camino mínimo a n_i es $D[n_i]$

Algoritmo de Dijkstra: Demostración de optimalidad (cont.)

- **Paso de inducción: b)**
  - en cada iteración del bucle, se actualizan las distancias de los nodos que no pertenecen a Y. Sean n_i el nodo que se acaba de añadir a Y, y $n_k \notin Y$.
  - Pueden ocurrir dos casos. Si el camino especial más corto (respecto a $Y \cup \{n_i\}$) no pasa por n_i, $D[n_k]$ no varía
  - En caso contrario, $D[n_k]$ será mayor o igual al nuevo camino especial más corto. Como en el algoritmo se asigna a $D[n_k]$ el mínimo entre el valor anterior y el nuevo (pasando por n_i), $D[n_k]$ tendrá el valor correcto
- Cuando se detenga el algoritmo, todos los nodos estarán en Y, y $D[2..n]$ contendrá los caminos más cortos desde n_1 hasta los demás nodos, con lo que el teorema queda demostrado
- ¿Cuál es la complejidad de este algoritmo?

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Problema de la mochila de N objetos

- Este problema es un clásico del diseño de algoritmos
- Veremos la versión más sencilla. Más adelante veremos otra técnica para resolver variantes más complejas
- Dados N objetos con pesos w_i y valores v_i, ambos positivos, y una mochila con capacidad de peso máxima W, debe diseñarse un algoritmo que encuentre la composición de la mochila cuyo valor sea máximo
- Los objetos son fraccionables y moldeables

Ejemplo: Problema de la mochila de N objetos

- Por ejemplo, tenemos tres objetos con distintos pesos y valores:

objeto	A	B	C
peso	5	10	20
valor	50	60	140

- ¿Cómo podemos seleccionar los productos para una mochila que tiene una capacidad máxima de 30?

Problema de la mochila de N objetos (cont.)

- Es un problema de optimización. Debemos encontrar las proporciones de los distintos productos en la mochila de forma que se maximice el valor de los objetos transportados
- La función objetivo por tanto consiste en maximizar $\sum_{i=1}^{n} x_i v_i$
- La función de factibilidad es la restricción $\sum_{i=1}^{n} x_i w_i \leq W$
- Se pueden fraccionar los elementos transportados. Por tanto, debemos determinar qué fracciones x_i de cada objeto optimizan la función objetivo, con $0 \leq x_i \leq 1$
- En el algoritmo voraz, los candidatos son los distintos objetos, y la solución es un vector $[x_1, ..., x_n]$ con la fracción de cada objeto que debemos incluir

Problema de la mochila de N objetos (cont.)

```
proc mochila(w[1..n],v[1..n],W,x[1..n])
   desde i ← 1 hasta n hacer
      x[i] ← 0
   fin desde
   peso ← 0
   mientras peso < W hacer
      i ← ⟨ el mejor objeto pendiente de considerar ⟩
      si peso + w[i]≤W entonces
         x[i] ← 1
         peso ← peso + w[i]
      si no
         x[i] ← (W - peso) / w[i]
         peso ← W
      fin si
   fin mientras
fin proc
```

Problema de la mochila de N objetos (cont.)

- La situación es más sencilla cuando todos los objetos caben en la mochila, $\sum_{i=1}^{n} w_i \leq W$, pero esto no tiene interés algorítmico
- El caso interesante se produce cuando $\sum_{i=1}^{n} x_i w_i > W$
- La estrategia general del algoritmo voraz consiste en seleccionar en cada paso un objeto y poner la mayor fracción posible de dicho objeto en la mochila
- Por tanto, habrá $j < n$ objetos tales que $x_1 = x_2 = ... = x_{j-1} = 1$, $x_j < 1$ y $x_{j+1} = ... = x_n = 0$
- Debemos determinar la función de selección de los objetos para encontrar la solución óptima
- Podemos utilizar tres funciones de selección:
  - Seleccionar primero el objeto más valioso
  - Seleccionar primero el objeto de menor peso
  - Seleccionar primero el objeto con mayor valor por unidad de peso

Ejemplo: Problema de la mochila de N objetos (cont.)

- En nuestro ejemplo

objeto	A	B	C
peso	5	10	20
valor	50	60	140

tenemos los siguientes resultados:

	A	B	C	total
Max v_i	0	1	1	200
Min w_i	1	1	0.75	215
Max v_i/w_i	1	0.5	1	220

- Por tanto, parece más interesante maximizar el valor por unidad de peso

Problema de la mochila de N objetos: Demostración de optimalidad

- Tenemos que demostrar que esta solución es la óptima:

  Teorema: Si se seleccionan los objetos por orden decreciente de v_i/w_i, entonces el algoritmo de la mochila encuentra una solución óptima
- Supongamos que los objetos están ordenados en orden decreciente de v_i/w_i, y sea $X = (x_1, ..., x_n)$ una solución proporcionada por este algoritmo
- Si $x_i = 1$ para todo i, la solución es óptima (podemos poner todos los objetos en la mochila)
- En caso contrario, por la forma en que funciona el algoritmo, $\sum_{i=1}^{n} x_i w_i = W$, y sea $V(X) = \sum_{i=1}^{n} x_i v_i$ el valor de la solución X
- Debemos demostrar que $V(X) \geq V(Y)$ para cualquier solución factible Y

Problema de la mochila de N objetos: Demostración de optimalidad (cont.)

- Sea $Y = (y_1, ..., y_n)$ cualquier solución factible. Por tanto, $\sum_{i=1}^{n} y_i w_i \leq W$, y $V(Y) = \sum_{i=1}^{n} y_i v_i$ es el valor de la solución Y
- Se verifica que $V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$
- Sea $j < n$ el índice del primer elemento tal que $x_j < 1$:
  - Cuando $i < j$, $x_i = 1$ y por tanto $x_i - y_i \geq 0$, y $v_i/w_i \geq v_j/w_j$
  - Cuando $i > j$, $x_i = 0$ y por tanto $x_i - y_i \leq 0$, y $v_i/w_i \leq v_j/w_j$
  - Si $i = j$, $v_i/w_i = v_j/w_j$
- En todos los casos, $(x_i - y_i) v_i/w_i \geq (x_i - y_i) v_j/w_j$. Por tanto,

$$V(X) - V(Y) \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$$

con lo que queda demostrado el teorema

Problema de la mochila 0-1

- Ahora supongamos que los objetos no son fraccionables (este problema se denomina de la mochila 0-1)
- En nuestro ejemplo,

objeto	A	B	C
peso	5	10	20
valor	50	60	140

el algoritmo voraz con la función de selección que se ha visto producirá el siguiente resultado:

	A	B	C	total
Max v_i/w_i	1	0	1	190

- Sin embargo, la siguiente solución es mejor:

A	B	C	total
0	1	1	200

- Más adelante veremos técnicas para solucionar este problema

Esquemas algorítmicos. Algoritmos voraces

1. Características generales de los algoritmos voraces
2. Ejemplo: Devolución del cambio
3. Árbol de recubrimiento mínimo. Algoritmo de Prim
4. Árbol de recubrimiento mínimo. Algoritmo de Kruskal
5. Caminos mínimos. Algoritmo de Dijkstra
6. El problema de la mochila de N objetos
7. Caballo de ajedrez

Caballo de ajedrez

- [GV00] Dado un tablero de ajedrez y una casilla inicial, queremos decidir si es posible que un caballo recorra todos y cada uno de los escaques sin repetir ninguno.
- Un posible algoritmo voraz decide en cada iteración la casilla desde la cual se domina el **menor** número posible de casillas no visitadas

a) Implementar este algoritmo para un tamaño de tablero NxN y una casilla inicial (x,y)

b) Buscar, utilizando este algoritmo, todas las casillas iniciales y tamaños de tablero para los que el algoritmo encuentra solución

c) Encontrar el patrón general de las soluciones del recorrido del caballo

Caballo de ajedrez (cont.)

- Necesitamos una estructura de datos para representar el recorrido que va haciendo el caballo, con varios objetivos:
  - proporcionarlo como salida del algoritmo
  - saber en cada paso qué escaques han sido visitados anteriormente para poder aplicar las funciones de selección y factibilidad
- Utilizaremos para representar el tablero una matriz cuadrada de enteros, inicialmente rellena con ceros.
- Cuando el algoritmo se detenga, este tablero contendrá números positivos que indicarán el orden que ha seguido el caballo en sus movimientos desde la posición inicial (x,y) recibida como entrada

Caballo de ajedrez (cont.)

```
//T es el tablero de salida
//(x,y) es la posicion inicial
fun caballo(T[1..N,1..N],x,y)
   desde i ← 1 hasta N hacer
      desde j ← 1 hasta N hacer
         T[i,j] ← 0
      fin desde
   fin desde
   desde i ← 1 hasta N*N hacer
      T[x,y] ← i
      si ¬ nuevo_movimiento(T,x,y) Y i<N*N-1 entonces
         devolver falso
      fin si
   fin desde
   devolver cierto
fin fun
```

Caballo de ajedrez (cont.)

- La función `nuevo_movimiento` obtiene en (x,y) las coordenadas del siguiente movimiento a realizar
- devuelve *falso* si no se puede realizar ningún movimiento

```
fun nuevo_movimiento(T[1..N,1..N],x,y)
  min_accesibles ← 9
  desde i ← 1 hasta 8 hacer
    si salto(T,i,x,y,nx,ny) entonces
      accesibles ← cuenta(T,nx,ny)
      si accesibles > 0 Y accesibles < min_accesibles entonces
        min_accesibles ← accesibles
        solx ← nx ; soly ← ny
      fin si
    fin si
  fin desde
  x ← solx ; y ← soly
  devolver min_accesibles < 9
fin fun
```

Caballo de ajedrez (cont.)

- Un caballo de ajedrez puede realizar hasta 8 movimientos desde una casilla determinada:

	2		3	
1				4
		C		
8				5
	7		6	

- Si numeramos los distintos movimientos posibles como se indica en el tablero, la función `salto` determina si se puede mover en cada una de estas posiciones, así como la casilla de destino del movimiento

Caballo de ajedrez (cont.)

- La función `salto` devuelve en (nx,ny) las coordenadas de la posición de destino del movimiento i-ésimo desde (x,y)
- devuelve *falso* si se no puede mover el caballo a la posición i-ésima

fun salto(T[1..N,1..N],i,x,y,nx,ny)
  seleccionar i
    1 **:**
      nx $\leftarrow$ x-2 ; ny $\leftarrow$ y+1
    2 **:**
      nx $\leftarrow$ x-1 ; ny $\leftarrow$ y+2
    ...
    8 **:**
      nx $\leftarrow$ x-2 ; ny $\leftarrow$ y-1
  fin seleccionar
  devolver ($1 \leq$ nx $\leq$ N **Y** $1 \leq$ ny $\leq$ N **Y** T[nx,ny]=0)
fin fun

Caballo de ajedrez (cont.)

- Por último, `cuenta` devuelve el número de posiciones válidas a las que puede moverse el caballo desde la posición (x,y)

```
fun cuenta(T[1..N,1..N],x,y)
  c ← 0
  desde j ← 1 hasta 8 hacer
    si salto(T,j,x,y,nx,ny) entonces
      c ← c + 1
    fin si
  fin desde
  devolver c
fin fun
```

Caballo de ajedrez (cont.)

- Para resolver b), podemos implementar un programa que utilice el algoritmo anterior con distintos tamaños de tablero y posiciones iniciales
- Después de hacer esto, podemos llegar a las siguientes conclusiones:
  - Para N=4 el problema no tiene solución
  - Para N>4 y par, el problema tiene solución para cualquier casilla inicial
  - Para N>4 impar, el problema tiene solución para las casillas iniciales (x,y) tales que x+y sea par
- Sin embargo, hay algunos casos en los que el algoritmo no ha encontrado solución
- Por ejemplo, para N=5 y (x,y)=(5,3) este algoritmo no encuentra solución

Caballo de ajedrez (cont.)

- Pero sí la encuentra para las casillas iniciales (1,3), (3,1) y (3,5), que son simétricas a la anterior

		S		
S				N
		S		

- ¿A qué se debe este comportamiento?
- Por tanto, este algoritmo **no** funciona para todos los casos
- Más adelante veremos la técnica de vuelta atrás que nos permitirá resolver este problema para todos los casos