ESQUEMA VORAZ

- INTRODUCCIÓN
- BASES DEL ESQUEMA VORAZ Y ESQUEMA GENERAL
- MOCHILA 0/1
- BÚSQUEDA DEL ÁRBOL DE RECUBRIMIENTO DE COSTE MÍNIMO: ALGORITMO DE KRUSKAL Y ALGORITMO DE PRIM
- CAMINOS MÍNIMOS: ALGORITMO DE DIJKSTRA
- HEURÍSTICAS VORACES

INTRODUCCIÓN

Ámbito de aplicación → idem Programación Dinámica y Backtracking

esto es,

Resolver un problema de OPTIMIZACIÓN:

- **sujeto a unas RESTRICCIONES,**
- **con una FUNCIÓN OBJETIVO conocida y explícita, y**
- **donde la estrategia de solución se basa también en la TOMA SECUENCIAL DE DECISIONES.**

INTRODUCCIÓN

El esquema VORAZ ofrece menos dificultad para generar el algoritmo solución.

Razones:

- **MIOPE, ya que a la hora de tomar una decisión no tiene en cuenta los efectos que esa decisión genera.**
- **TOZUDO, ya que una vez tomada esa decisión (sea la que fuera) ya no ofrecen la posibilidad de reconsiderarla.**

INTRODUCCIÓN

ESTRATEGIA:

- **En cada decisión, a partir de la información disponible, eligen la MEJOR OPCIÓN.**
- **Lo hacen de acuerdo a un CRITERIO TAN RAZONABLE como sea posible.**
- **¡¡PROBLEMA!!: esa opción mejor lo es de acuerdo a ese criterio razonable, pero no podemos garantizar que corresponda a la óptima.**

INTRODUCCIÓN

VENTAJAS:

- **De fácil diseño e implementación**
- **Suelen ser algoritmos muy eficientes**

DESVENTAJAS:

- **Por las limitaciones expuestas, son inservibles en algunos casos.**

EJEMPLO: DEVOLVER CAMBIO

Supongamos que somos el empleado de una tienda y que tenemos que devolver un cambio a un cliente por importe de N €.

Para ello disponemos de tres tipos de monedas:

- **1€,**
- **2€ y**
- **5€.**

Se supone que en caja hay un número inagotable de monedas de cada tipo.

Nos hemos fijado el objetivo de MINIMIZAR el número de monedas incluidas en el cambio.

EJEMPLO: DEVOLVER CAMBIO

La solución se plantea como una SECUENCIA DE DECISIONES.

FUNCIÓN OBJETIVO: utilizar el MENOR número de monedas para componer la cantidad N euros.

Y las condiciones de FACTIBILIDAD son:

- **En todo momento la cantidad "construida" debe ser menor o igual que N.**
- **Y finalmente, claro está, debe de ser N.**

EJEMPLO: DEVOLVER CAMBIO

Estrategia Voraz.- elegir ¿.....?

Ejemplo: N = 18€ y Tipos de monedas = {1€, 2€, 5€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	18

EJEMPLO: DEVOLVER CAMBIO

Estrategia Voraz.- elegir en cada etapa el tipo de moneda de mayor valor.

Ejemplo: N = 18€ y Tipos de monedas = {1€, 2€, 5€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	18

EJEMPLO: DEVOLVER CAMBIO

Ejemplo: N = 18€ y Tipos de monedas = {1€, 2€, 5€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	18
1	5	3 monedas de 5 euros	3

EJEMPLO: DEVOLVER CAMBIO

Ejemplo: N = 18€ y Tipos de monedas = {1€, 2€, 5€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	18
1	5	3 de 5	3
2	2	3 de 5, 1 de 2	1

EJEMPLO: DEVOLVER CAMBIO

Ejemplo: N = 18€ y Tipos de monedas = {1€, 2€, 5€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	18
1	5	3 de 5	3
2	2	3 de 5, 1 de 2	1
3	1	3 de 5, 1 de 2, 1 de 1	0

SOLUCIÓN ÓPTIMA: 5 monedas (3 de 5€, 1 de 2€ y 1 de 1€)

No puede generalizarse, pues depende del sistema monetario.

EJEMPLO: DEVOLVER CAMBIO

Ejemplo: N = 36 € y Tipos de monedas = {1€, 5€, 12€, 25€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	36
1	25	1 de 25	11
2	12	1 de 25, 0 de 12	11
3	5	1 de 25, 0 de 12, 2 de 5	1
4	1	1 de 25, 0 de 12, 2 de 5, 1 de 1	0

Solución: 4 monedas (1 de 25€, 0 de 12€, 2 de 5€, 1 de 1€)

¡¡ SOLUCIÓN NO ÓPTIMA !!

SÓLUCIÓN ÓPTIMA: 3 monedas (3 de 12€)

EJEMPLO: DEVOLVER CAMBIO

Ejemplo: N = 36 € y Tipos de monedas = {5€, 12€, 25€}

Etapa	Tipo de moneda seleccionada	Número de monedas utilizadas	Cantidad a devolver
inicial	--	0	36
1	25	1 de 25	11
2	12	1 de 25, 0 de 12	11
3	5	1 de 25, 0 de 12, 2 de 5	1

¡¡ NO HAY SOLUCIÓN !!

SÓLUCIÓN ÓPTIMA: 3 monedas (3 de 12€)

EJEMPLO: DEVOLVER CAMBIO

Demostrar con generalidad si la solución voraz en el problema de devolver el cambio es óptima o no, es un problema complejo que no abordaremos aquí.

Hemos visto que dependiendo de los tipos de monedas disponibles el algoritmo voraz puede no funcionar de la manera esperada ya que:

- **en algunos casos puede seleccionar un conjunto de monedas que no sea óptimo y**
- **en otros casos puede no llegar a encontrar la solución, aún cuando ésta sí exista.**

Ocurre lo mismo si el suministro de alguna de los tipos de monedas es limitado.

EJEMPLO: DEVOLVER CAMBIO

- Funcionamiento por ETAPAS (BUCLE).
- En cada etapa (iteración) SELECCIONAMOS el tipo de moneda de MAYOR valor.
- Una vez seleccionado un tipo de moneda:
  - SE AÑADE A LA SOLUCIÓN siempre y cuando su valor sea menor o igual que la cantidad a devolver o
  - SE RECHAZA si su valor es mayor ya que sino la solución no cumpliría las restricciones del problema (FACTIBILIDAD).
- Una vez seleccionado un tipo de moneda, pase o no a formar parte de la solución, no se vuelve a seleccionar en etapas (iteraciones) posteriores (DECISIONES IRREVOCABLES).
- Se genera una ÚNICA secuencia de decisiones.

EJEMPLO: DEVOLVER CAMBIO


```
Función Devolver_cambio ( N : entero) retorna ( S : conjunto de monedas)
    C = { 1, 5, 12, 25 }              /* C es el sistema monetario disponible */
    S  = ∅                            /* S es el conjunto que contendrá la solución */
    cantidad = N                      /* cantidad es la cantidad que resta por cubrir */
    mientras ( cantidad ≠ 0 ∧ C ≠ ∅ ) hacer
             x ← el mayor elemento de C
             C = C - { x }
             si cantidad ≥ x entonces
                    y = cantidad / x   /* división entera */
                    S ← S ∪ { y monedas de valor x }
                    cantidad = cantidad - x * y
             fsi
    fmientras
    si cantidad = 0 entonces retorna S  sino retorna ∅  fsi
ffuncion
```

BASES DEL ESQUEMA VORAZ

La estrategia voraz se apoya en unas bases que podemos resumir del siguiente modo:

- **En cada decisión debemos elegir de UN CONJUNTO DE CANDIDATOS en base a algún criterio razonable.**
- **Una función de SELECCIÓN de los candidatos determinará en cada momento cuál es el mejor candidato posible.**
- **Utilizamos dos conjuntos SOLUCIÓN (S) y CANDIDATOS (C).**
- **A medida que vamos tomando decisiones, el conjunto de CANDIDATOS se reduce y el conjunto SOLUCIÓN puede ir aumentando (aumentará si es factible; sino, no).**

BASES DEL ESQUEMA VORAZ

- **Una función FACTIBLE nos permitirá determinar si la decisión en curso (MEJOR CANDIDATO ACTUAL) incorporada a la solución que llevamos hasta ese momento (solución parcial) hace que se sigan cumpliendo las restricciones del problema. Si la respuesta es SI, la decisión actual se añade a la solución; si la respuesta es NO, no se añade a la solución.**

- **Una función ES_SOLUCIÓN nos permitirá determinar en cada momento si el conjunto solución resuelve el problema planteado.**

EL ESQUEMA GENERAL VORAZ


```
Función Voraz ( C : conjunto ) retorna ( S : conjunto )

   S = ∅
   mientras ( ¬ ES_SOLUCIÓN(S) ∧ C ≠ ∅ )   hacer
          x ← SELECCIÓN (C)                /* seleccionamos el mejor candidato */
          C  = C – {x}                     /* lo eliminamos del conjunto solución  */
          si  FACTIBLE (S ∪ {x}) entonces  /*  si es factible incorporarlo a la solución */
                 S ← S ∪ {x}               /* lo añadimos al conjunto solución */
          fsi
    fmientras

    si ES_SOLUCIÓN (S) entonces retorna  S sino retorna  ∅  fsi
ffuncion
```

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz ( x : T1 ) retorna ( y : T2 )
    var z : T´1, S : T2, decision : T3 fvar
    z = prepara (x);
    S = solucion_vacía;
    mientras ( ¬es_solucion (S) ∧ z ≠ ∅ ) hacer
            decision = selecciona_siguiente (z);
            z = elimina(z, decision);
            si factible (S, decision) entonces
                    S = añade (S, decision);
            fsi
    fmientras
    si es_solucion (S) entonces retorna S  sino retorna solucion_vacía  fsi
ffuncion
```

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz ( x : T1 ) retorna ( y : T2 )
   var z : T´1, S : T2, decision : T3 fvar
   z = prepara (x);
   S = solucion_vacía;
   mientras ¬es_solucion (S) ∧  z ≠ ∅  hacer
    …
   finmientras
   …
   finfuncion
```

Funcion prepara (x : T1) retorna (z : T´1)

Comportamiento: Devuelve una estructura transformada de los datos de entrada x, diseñada para facilitar la aplicación del criterio de selección adoptado.

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz (x : T1) retorna (y : T2)
   var z : T´1, S : T2, decision : T3 fvar
   z = prepara (x);
   S = solucion_vacía;
   mientras ¬es_solucion (S) ∧  z ≠ ∅  hacer
   …
  finmientras
  …
finfuncion
```

Funcion es_solucion (S : T2) retorna (b : booleano)

Comportamiento: Devuelve cierto si la solución obtenida es solución del problema planteado; falso, en caso contrario.

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz (x:T1) retorna (y:T2)
  var z : T´1, S : T2, decisión : T3 fvar
  z = prepara (x);
  S = solucion_vacía;
  mientras ¬es_solución (S) ∧ z ≠ ∅ hacer
     decision = selecciona_siguiente (z);
  ...
  finmientras
  ...
finfuncion
```

Función selecciona_siguiente (z : T´1) retorna (y : T3)

Comportamiento: Selecciona uno de los elementos de z según cierto criterio.

EL ESQUEMA GENERAL VORAZ DETALLADO

Función Voraz (x:T1) retorna (y:T2)

…

mientras ¬es_solución (S) $\wedge$ z $\neq \varnothing$ hacer

decisión = selecciona_siguiente (z);

z = elimina (z, decisión);

…

finmientras

…

finfuncion

Funcion elimina (z : T´1; decisión : T3) retorna (z : T´1)

Comportamiento: Actualiza el conjunto z de candidatos a considerar, devolviendo un nuevo conjunto al que ya no pertenece la decisión elegida.

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz (x:T1) retorna (y:T2)
…
mientras ¬es_solución (S) ∧  z ≠ ∅  hacer
            …
            si factible (S, decisión) entonces
                      S = añade (S, decisión);
           fsi
finmientras
…
finfuncion
```

Funcion factible (S: T2, decisión : T3) retorna (b : booleano)

Comportamiento: Devuelve cierto si la inclusión de la decisión en la solución parcial conduce a una solución que no viola las restricciones, es decir, si progresa como una solución factible; falso, en caso contrario.

EL ESQUEMA GENERAL VORAZ DETALLADO


```
Función Voraz (x:T1) retorna (y:T2)
…
mientras ¬es_solución (S) ∧ z ≠ ∅ hacer
        …
        si factible (S, decisión) entonces
                S = añade (S, decisión);
        fsi
finmientras
…
finfuncion
```

Funcion añade (S : T2, decisión : T3) retorna (S : T2)

Comportamiento: Devuelve la combinación de la decisión con la solución parcial anterior.

MOCHILA [0,1]

Se dispone de n objetos con sus correspondientes pesos (P) y beneficios (B) y tenemos una mochila con una capacidad C. Se trata de determinar qué objetos y en qué proporción se introducen en la mochila con la finalidad de conseguir el máximo valor.

Solución como secuencia de decisiones:

$$X_1 X_2 \ldots X_n$$

donde X_i es la decisión tomada para el objeto i-ésimo. Su valor indicará la porción del objeto que se introduce, esto es X_i pertenece a [0,1].

MOCHILA [0,1]

De tal forma que la secuencia óptima de decisiones será aquella

$$X_1 X_2 \ldots X_n$$

tal que

$$\max \sum_{i=1}^{n} x_i B_i$$

$$\text{sujeto a} \sum_{i=1}^{n} x_i P_i \leq C$$

MOCHILA [0,1]

¿ CRITERIO/S DE SELECCIÓN ?

MOCHILA [0,1]

N=3 C=20 P[1..3]={18, 15, 10} B[1..3]={25, 24, 15}

Criterio 1: MÁXIMO VALOR

Etapa	Objeto seleccionado	(X_1, X_2, X_3)	Peso máximo	Valor total
inicial	--	(0, 0, 0)	20	0
1	1	(1, 0, 0)	2	25
2	2	(1, 2/15, 0)	0	28.2

MOCHILA [0,1]

N=3 C=20 P[1..3]={18, 15, 10} B[1..3]={25, 24, 15}

Criterio 2: MÍNIMO PESO

Etapa	Objeto seleccionado	(X_1, X_2, X_3)	Peso máximo	Valor total
inicial	--	(0, 0, 0)	20	0
1	3	(0, 0, 1)	10	15
2	2	(0, 2/3, 1)	0	31

MOCHILA [0,1]

N=3 C=20 P[1..3]={18, 15, 10} B[1..3]={25, 24, 15}

Criterio 3: MAYOR Bi/Pi (Ratio)

Ratio [1..3]={1.39, 1.6, 1.5}

Etapa	Objeto seleccionado	(X_1, X_2, X_3)	Peso máximo	Valor total
inicial	--	(0, 0, 0)	20	0
1	2	(0, 1, 0)	5	24
2	3	(0, 1, 1/2)	0	31.5

MOCHILA [0,1]

Teorema: **Si se seleccionan los objetos por orden decreciente de B_i/P_i, entonces el algoritmo voraz de la Mochila [0,1] encuentra la solución óptima.**

MOCHILA [0,1]


```
Función Mochila_con_particion (N, C:entero, P[1..N], B[1..N]: vector de enteros) retorna ( Bmax : entero, X[1..N]: vector de reales )
var capacidad, beneficio, i : entero; X[1..N]:vector de reales fvar
capacidad = C;
beneficio=0;
para i=1 hasta N hacer   X[i]=0.0   fpara
mientras capacidad ≠ 0 hacer                              /* bucle voraz */
        i ← posición de aquel objeto no seleccionado aún cuya ratio Bi/Pi sea mayor
        si P[i] ≤ capacidad  entonces
                  X[i]=1;
                  capacidad = capacidad - P[i];
                  beneficio = beneficio + B[i];
         sino
                X[i] = capacidad / P[i];
                capacidad = 0;
                beneficio = beneficio + B[i] * X[i];
        fsi
fmientras
retorna (beneficio, X[1..N]);
ffunción
```

MOCHILA {0,1}

Ejemplo utilizado en Programación Dinámica y Backtracking.-

N=3 C=15 P[1..3]={5, 6, 9} B=[1..3]={ 24, 40, 38 }

Criterio 3: MAYOR B_i/P_i

Solución → $(X_1, X_2, X_3) = (1, 1, 0)$ Beneficio: 64

¡¡ SOLUCIÓN NO ÓPTIMA!!

Criterio 1: MÁXIMO VALOR

Solución → $(X_1, X_2, X_3) = (0, 1, 1)$ Beneficio: 78

¡¡ SOLUCIÓN ÓPTIMA!!

ALGORITMOS VORACES QUE PRODUCEN LA SOLUCIÓN ÓPTIMA

A pesar de los ejemplos vistos (devolver el cambio y mochila), existen algunas familias de problemas para las que existe una solución voraz óptima.

Estudiaremos dos problemas ligados a grafos:

- **la búsqueda del árbol de recubrimiento de coste mínimo de un grafo (algoritmo de Kruskal y algoritmo de Prim) y**
- **la búsqueda del camino más corto desde un único nodo origen hasta los demás nodos de un grafo (algoritmo de Dijkstra).**

BÚSQUEDA DEL ÁRBOL DE RECUBRIMIENTO DE COSTE MÍNIMO

Sea G = <N, A> un grafo conexo no dirigido, donde N es el conjunto de nodos y A es el conjunto de ejes.

Además, existe una función L: $A \rightarrow R^+$ denominada longitud o coste de un eje.

El problema consiste en hallar un grafo parcial de G,

G' = <N, T>

donde $T \subset A$, que satisfaga las siguientes condiciones:

- **Los ejes de T deben mantener conectados todos los nodos de G.**
- **La suma de las longitudes de los ejes de T debe ser mínima.**

- **El grafo G´ SIEMPRE existe dado que G es finito y conexo.**

- **G´ debe ser un ÁRBOL (POR TEORÍA DE GRAFOS)**

  - **Dado que G´ debe ser conexo debe tener al menos n-1 ejes, supuesto que el número de nodos de G es n.**

  - **Si G´ tuviera más de n-1 ejes, eso querría decir que tendría al menos un ciclo, lo que implicaría que podríamos eliminar un eje de ese ciclo y G´ seguiría siendo conexo, lo que daría lugar a una solución de menor coste.**

CONCLUSIÓN.- G´ debe de ser un ÁRBOL → n-1 ejes.

$$| T | = n-1$$

ALGORITMO DE KRUSKAL

Candidatos: Los ejes de A

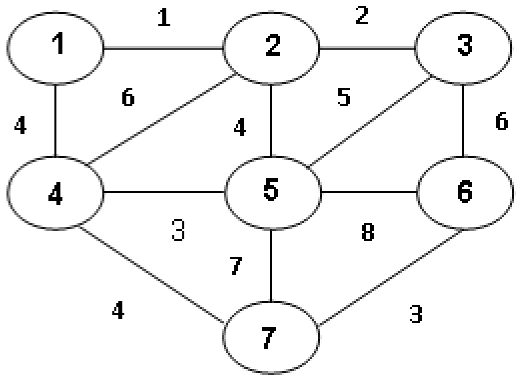
Estrategia: La idea es que todos los nodos están inicialmente aislados (hay n componentes conexas inicialmente) y vamos añadiendo sucesivamente ejes, de modo que en cada selección el número de componentes conexas se reduce en una unidad.

Candidato más prometedor: Eje de A con MENOR longitud

Factibilidad: El eje elegido debe de tener los extremos en componentes conexas diferentes, así nos aseguramos de que no forma ciclo con los que ya han sido seleccionados anteriormente.

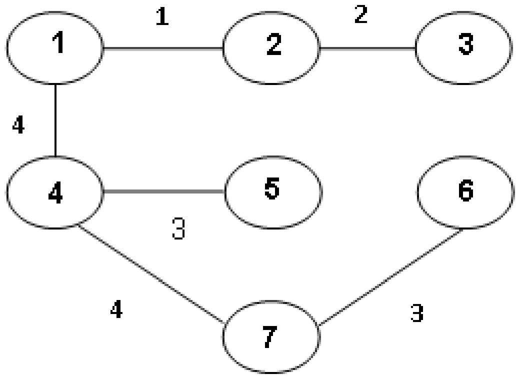
ALGORITMO DE KRUSKAL

Veamos el funcionamiento de la estrategia de Kruskal con el siguiente ejemplo:



ALGORITMO DE KRUSKAL

Etapa	Arista seleccionada	A/R	Componentes conexas
inicial	--	--	{1} {2} {3} {4} {5} {6} {7}
1	{1,2}	A	{1,2} {3} {4} {5} {6} {7}
2	{2,3}	A	{1,2,3} {4} {5} {6} {7}
3	{4,5}	A	{1,2,3} {4,5} {6} {7}
4	{6,7}	A	{1,2,3} {4,5} {6,7}
5	{1,4}	A	{1,2,3,4,5} {6,7}
6	{2,5}	R	{1,2,3,4,5} {6,7}
7	{4,7}	A	{1,2,3,4,5,6,7}



ALGORITMO DE KRUSKAL


```
Función Kruskal ( G = <N,A> : grafo; L : A → R⁺ ) retorna ( T : conjunto de ejes )

        n = | N |
        T = ∅               /* conjunto de ejes que forman la solución */
        Inicializar las n componentes conexas
        mientras | T | < n - 1 hacer    /* bucle voraz */
                e ← eje {u,v} de menor longitud de los aún no seleccionados
                comp_u  = buscar(u)
                comp_v  = buscar(v)
                si (comp_u ≠ comp_v) entonces
                                fusionar (comp_u, comp_v)
                                T = T ∪ {e}
                fsi
         fmientras
         retornar T
ffuncion
```

ALGORITMO DE KRUSKAL

Para diseñar el algoritmo hemos necesitado dos operaciones auxiliares:

- **buscar(x): devuelve la componente conexa a la que pertenece el nodo x**
- **fusionar(A,B): fusiona las componentes conexas A y B como consecuencia de la adición de un nuevo eje.**

Observar que a lo largo del proceso existirán varias componentes conexas.

Cada componente conexa es en sí misma un árbol de expansión mínimo para el conjunto de nodos que la forman.

Teorema: El algoritmo de Kruskal halla un árbol de expansión mínimo.

ALGORITMO DE PRIM

Candidatos: Los ejes de A

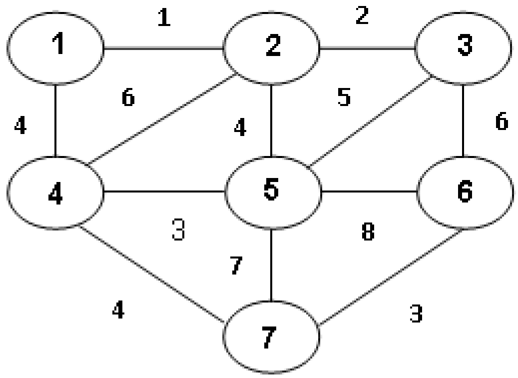
Estrategia: Partiendo de un nodo cualquiera vamos creando una componente conexa cada vez mayor, haciendo que cada eje elegido extienda esa componente a un nodo nuevo.

Candidato más prometedor: Eje de A con MENOR longitud.

Factibilidad: El eje a añadir debe tener un extremo dentro y otro fuera de la componente conexa, para aseguramos de que no forma ciclo con los que ya están en la solución.

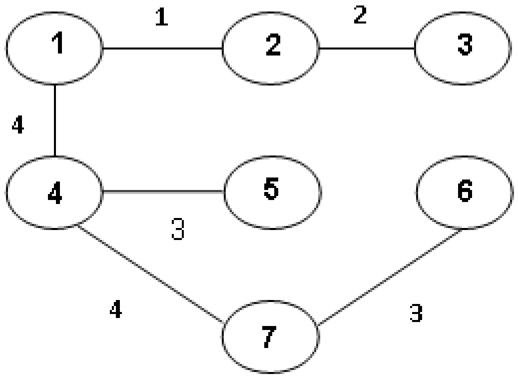
ALGORITMO DE PRIM

Veamos el funcionamiento de la estrategia de Prim con el ejemplo anterior:



ALGORITMO DE PRIM

Etapa	Arista seleccionada	B
inicial	--	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{6,7}	{1,2,3,4,5,6,7}



ALGORITMO DE PRIM

Función PRIM (G = <N,A> : grafo; L : A $\rightarrow$ R+) retorna (T : conjunto de ejes)

```
    B ← un nodo arbitrario de N
    T = ∅    /* conjunto de ejes que forman la solución */
    mientras B ≠ N  hacer   /* bucle voraz */
        Buscar e = {u,v} de longitud mínima, tal que u∈B  y  v∉B
        T = T ∪ {e}
        B = B ∪ {v}
    fmientras
    retorna T
ffunción
```

ALGORITMO DE PRIM (DETALLADO)

Mas_proximo[i]

almacena en cada paso el nodo más próximo a i (i NO pertenece a B) entre los pertenecientes a la componente conexa (pertenecientes a B) en ese momento.

Dist_min[i]

almacena en cada paso la distancia del nodo i (i NO pertenece a B) al nodo más próximo entre los pertenecientes a la componente conexa (pertenecientes a B) en ese momento.

Cuando i pasa a pertenecer a B, Dist_min[i] toma valor -1.

ALGORITMO DE PRIM (DETALLADO)


```
Función PRIM ( L[1..n, 1..n])  retorna (T : conjunto de ejes )
/* inicialización */
B = {nodo 1}
T = ∅   /* conjunto de ejes que forman la solución */
para i = 2   hasta  n   hacer
            Mas_proximo[i] = 1
            Dist_min[i] = L[i][1]
fpara
            …
```

ALGORITMO DE PRIM (DETALLADO)


```
…
Repetir  n - 1 veces    /*  bucle voraz */
    min  = ∞
    para i = 2   hasta  n   Hacer
            si  0 ≤ Dist_min[ i ] < min   entonces
                        min = Dist_min[i]
                        k = i
             fsi
   fpara
   T ← T ∪  { Mas_proximo[k], k }
   Dist_min[k] = -1       / *  se añade k a B */
   para i=2  hasta  n   hacer
           si  L[i][k] < Dist_min[i]   entonces
                        Dist_min[i] = L[i][k]
                        Mas-proximo[i] = k
           fsi
   fpara
frepetir
retorna T
ffunción
```

ALGORITMO DE PRIM

A lo largo del proceso, existirá una componente conexa creciente a la que se van añadiendo sucesivamente ejes.

En todo momento esa componente conexa es en sí misma un árbol de expansión mínima para el conjunto de nodos que la forman.

Teorema: El algoritmo de Prim halla un árbol de recubrimiento mínimo.

ÓRDENES DE COMPLEJIDAD DE KRUSKAL Y PRIM

Disponemos de 2 algoritmos que resuelven el mismo problema

¿Cuál usamos?

Prim $\in O(n^2)$ siendo n el número de nodos.

Kruskal $\in O(a \log n)$, siendo a el número de ejes del grafo.

Grafo completo: el número de ejes es $n(n-1)/2 \Rightarrow$ Kruskal $\in O(n^2 \log n)$,

Grafo con número mínimo de ejes para ser conexo, esto es, con n-1 ejes $\Rightarrow$ Kruskal $\in O(n \log n)$.

La preferencia por uno u otro depende de la "densidad" del grafo, de tal forma que a medida que ésta crece, el algoritmo de Kruskal va perdiendo interés, y lo va ganando el de Prim.

CAMINOS MÍNIMOS

Sea un grafo dirigido G = <N,A>.

Cada arco (i,j) de A tiene una longitud L[i][j] $\geq$ 0.

Por convenio, asumimos que L[i][j] = ∞ si (i,j) $\notin$ A.

El problema consiste en determinar la longitud del camino mínimo que va desde un nodo cualquiera del grafo hasta cada uno de los restantes nodos.

ALGORITMO DE DIJKSTRA

El algoritmo que permite resolver este problema es el algoritmo de Dijkstra, y se basa en:

- **Mantener a lo largo del proceso dos conjuntos S y C, de tal forma que S contenga los nodos para los que ya es conocida la longitud del camino mínimo desde el origen (nodos ya seleccionados). C contendrá los restantes nodos (nodos no seleccionados aún). Al inicio del proceso, S sólo contiene el nodo origen. En todo momento se cumple que $N = S \cup C$.**

- **Utilizar el concepto de camino especial: Decimos que un camino desde el origen hasta cualquier otro nodo es especial si todos los nodos intermedios de ese camino pertenecen a S.**

ALGORITMO DE DIJKSTRA

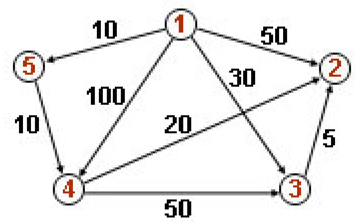
- **Mantener actualizado un vector D que contiene para cada nodo la longitud del camino especial más corta desde el origen.**

- **Seleccionar en cada etapa aquel nodo v de C cuya distancia desde el origen sea mínima y lo añadimos a S. El camino especial más corto hasta v es el más corto de todos los caminos posibles hasta v. (demostración por inducción).**

- **De esta forma, al acabar el proceso, todos los nodos estarán incluidos en S y, en consecuencia, D contendrá la solución del problema.**

- **Con estos elementos puede demostrarse que la estrategia propuesta por Dijkstra proporciona los caminos óptimos.**

ALGORITMO DE DIJKSTRA



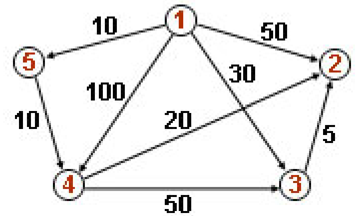
Inicialmente S = {1} y C = {2, 3, 4, 5}.

La composición inicial de D[2..5] = [50, 30, 100, 10] corresponde a las longitudes de los arcos que unen el nodo origen, el 1, con los restantes nodos.

Veamos a continuación cómo la estrategia voraz construye la solución.

Etapa	Nodo selec	S	C	D[2..5]	P[2..5]
inicial	--	{1}	{2, 3, 4, 5}	[50, 30, 100, 10]	[1, 1, 1, 1]
1	5	{1,5}	{2, 3, 4}	[50, 30, 20, 10]	[1, 1, 5, 1]
2	4	{1,5,4}	{2, 3}	[40, 30, 20, 10]	[4, 1, 5, 1]
3	3	{1,5,4,3}	{2}	[35, 30, 20, 10]	[3, 1, 5, 1]

ALGORITMO DE DIJKSTRA



D[2..5] = { 35, 30, 20, 10 }   P[2..5] = { 3, 1, 5, 1 }

Camino mínimo de 1 a 2: 1 → 3 → 2

Camino mínimo de 1 a 3: 1 → 3

Camino mínimo de 1 a 4: 1 → 5 → 4

Camino mínimo de 1 a 5: 1 →5

ALGORITMO DE DIJKSTRA


```
Función Dijkstra ( L[1..n][1..n] : matriz ) retorna ( D[2..n] : vector )
    C = { 2, 3, …, n }                        /* implícitamente S = { 1 } */
    para i  = 2  hasta   n  hacer
        D[ i ] = L[ 1 ][ i ]
        P[ i ] = 1
    fpara
    repetir   n-2 veces     /* bucle voraz */
            v ← aquel elemento de C que minimice D[v]
            C = C – {v}                       /* implícitamente  S = S ∪ { v } */
            para cada  w ∈ C   hacer
                    si D[w] > D[v] + L[v][w] entonces
                        D[w]=D[v] + L[v][w]
                        P[w]=v
                    fsi
            fpara
    frepetir
ffunción
```

ALGORITMO DE DIJKSTRA


```
Función Dijkstra (L[1..n][1..n]:matriz, ORIGEN:entero)   retorna (D[1..n]: vector)
    C = { 1, 2, 3, …, n } – ORIGEN              /* implícitamente S = { ORIGEN } */
    para i  = 1  hasta   n    Hacer
        D[i] = L[ ORIGEN ][ i ]
        P[i] = ORIGEN
    fpara
    repetir   n-2 veces      /* bucle voraz */
        v ← aquel elemento de C que minimice D[v]
        C = C – {v}                             /* implícitamente S = S ∪ {v} */
        para cada  w ∈ C   Hacer
            si D[w] > D[v] + L[v][w] entonces
                D[w]=D[v] + L[v][w]
                P[w]=v
            fsi
        fpara
    frepetir
ffunción
```

ALGORITMO DE DIJKSTRA

Teorema: El algoritmo de Dijkstra halla los caminos más cortos desde un único origen hasta los demás nodos de un grafo.

HEURÍSTICAS VORACES

Por ser, en general, simples, a veces se utilizan los algoritmos voraces como heurísticas (algoritmos de aproximación) para obtener soluciones subóptimas (cercanas a las óptimas) cuando esto es suficiente.

Existen problemas como el coloreado de grafo donde todos los algoritmos exactos conocidos emplean un tiempo exponencial. Como éstos no son utilizables para ejemplares de gran tamaño, sólo podemos emplear métodos aproximados.

En estos casos, un algoritmo voraz tiene la posibilidad, pero no la certeza, de encontrar la solución óptima.