Tabla de Contenidos

Programación dinámica

- Se emplea para resolver problemas de optimización
- Permite resolver problemas mediante una secuencia de decisiones (*como el esquema voraz*)
- Se producen varias secuencias de decisiones
  - Solo al final se sabe cuál es la mejor de ellas (*diferencia con esquema voraz*)
- Cuando se aplica, la solución a un problema de tamaño *n* se puede expresar en función de tamaño *n-1*
  - $f_N(n) = g(n) + f_{N-1}(n)$ (*método ascendente*)
- Basada en el principio de **Optimalidad de Bellman**

Ejemplo de aplicación

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 1 < k < n \\ 1 & \text{en otro caso} \end{cases}$$

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

Ejemplo base

- Problema de inversión económica en entidades bancarias
  - 4 unidades de dinero
  - 3 entidades bancarias
  - La inversión de la cantidad x_i en la entidad j produce una ganancia de $f_j(x_i)$ unidades de dinero
  - $f_j(x_i)$

	0	1	2	3	4
f1	0	2	5	6	7
f2	0	1	3	6	7
f3	0	1	4	5	8

Ejemplo base

- Solución al problema: **[a,b,c]** con a+b+c≤4
  - a,b,c unidades respectivamente en entidades I, II, III
- Si **[a*,b*,c*]** es una solución óptima para repartir **4** unidades en (I,II,III) entontes
  - **[a*,b*]** es una solución óptima para el subproblema de repartir **a*+b*** entre (I,II), o bien
  - **[a*,c*]** es una solución óptima para el subproblema de repartir **a*+c*** entre (I,III)
- Forma general de resolución
  - Resolver subproblemas en orden hasta llegar al original

Ejemplo base – Forma de la solución final

Problema original que puede tener diversas soluciones

A. 4 $\rightarrow$ (I, II)

B. 3 $\rightarrow$ (I, II) + 1 $\rightarrow$ (III)

C. 2 $\rightarrow$ (I, II) + 2 $\rightarrow$ (III)

D. 1 $\rightarrow$ (I, II) + 3 $\rightarrow$ (III)

E. 4 $\rightarrow$ (III)

Se obtienen 8 subproblemas que se pueden estudiar por separado

La solución óptima será la que produzca una mayor ganancia

Ejemplo base – Solución A (4→I,II)

a) 4 → (I) + 0 → (II) ganancia = 7
b) 3 → (I) + 1 → (II) ganancia = 7
c) 2 → (I) + 2 → (II) ganancia = 8
d) 1 → (I) + 3 → (II) ganancia = 8
e) 4 → (II) ganancia = 7

*gan A = **max** {gan(a),gan(b),gan(c),gan(d),gan(e)} = 8*

Solución elegida [2,2] que produce una ganancia de 8

	0	**1**	**2**	**3**	**4**
f1	0	2	5	6	7
f2	0	1	3	6	7
f3	0	1	4	5	8

Ejemplo base – Solución B (3→I,II + I→III)

Aparecen dos subproblemas

3 → (I) + 0 → (II)	ganancia = 6
2 → (I) + 1 → (II)	ganancia = 6
1 → (I) + 2 → (II)	ganancia = 5
0 → (I) + 3 → (II)	ganancia = 6

1 → (III)	ganancia = 1

	0	1	2	3	4
f1	0	2	5	6	7
f2	0	1	3	6	7
f3	0	1	4	5	8

Solución elegida
[3,0,1] que produce
una ganancia de 7

Ejemplo base – Solución C (2→I,II + 2→III)

Aparecen dos subproblemas

2 → (I) + 0 → (II)	ganancia = 5
1 → (I) + 1 → (II)	ganancia = 3
0 → (I) + 2 → (II)	ganancia = 3

2 → (III)	ganancia = 4

	0	1	2	3	4
f1	0	2	5	6	7
f2	0	1	3	6	7
f3	0	1	4	5	8

Solución elegida [2,0,2] que produce una ganancia de 9

Ejemplo base – Solución D (1→I,II + 3→III)

Aparecen dos subproblemas muy **triviales**

1 → (I) + 0 → (II) ganancia = 2

0 → (I) + 1 → (II) ganancia = 1

3 → (III) ganancia = 5

Ejemplo base – Solución E (4→III)

[0,0,4] ganancia = 8

	0	**1**	**2**	**3**	**4**
f1	0	2	5	6	7
f2	0	1	3	6	7
f3	0	1	4	5	8

Ejemplo base - generalización

- Tomar los datos de la tabla de soluciones triviales
- Plantear y resolver todos los subproblemas y aprovechar soluciones obtenidas en etapas anteriores
- Seguir haciendo agrupaciones hasta llegar al problema inicial

- *Se aborda la resolución de los subproblemas organizándolos en diversas etapas*

Ejemplo base - etapas

Etapas / Subetapas	Grupos de 1 entidad	Grupos de 2 entidades	Grupos de 3 entidades
1 Unidad	$1 \rightarrow I$ $1 \rightarrow II$ $1 \rightarrow III$		
2 Unidades	$2 \rightarrow I$ $2 \rightarrow II$ $2 \rightarrow III$		
3 Unidades	$3 \rightarrow I$ $3 \rightarrow II$ $3 \rightarrow III$		
4 unidades	$4 \rightarrow I$ $4 \rightarrow II$ $4 \rightarrow III$		Problema original

Programación dinámica

- No disponemos de principio de una fórmula de algoritmo general
- Procedimiento
  - Analizar tipos de problemas resolubles con PD
  - Deducir un esquema general de algoritmo dinámico
- Problemas que resolveremos
  - **Caminos óptimos en grafos multietápicos**
  - Mochila
  - Viajante de comercio
  - Árbol de búsqueda óptimo

Principio (POB)

- Toda subpolítica de una política óptima es también óptima
- Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve

Aplicación

- **Problema del camino más corto**
  - Conjunto de nodos unidos por aristas valoradas
  - Para ir desde el nodo **X** hasta el **Y** el mejor camino es **M(X,x_1,...,Y)**
  - Un subcamino de **M**, **K(T,x_h,...,Z)** debe ser el mejor camino de **X** a **Z**
  - Se cumple siempre que **T** y **Z** estén en **M**

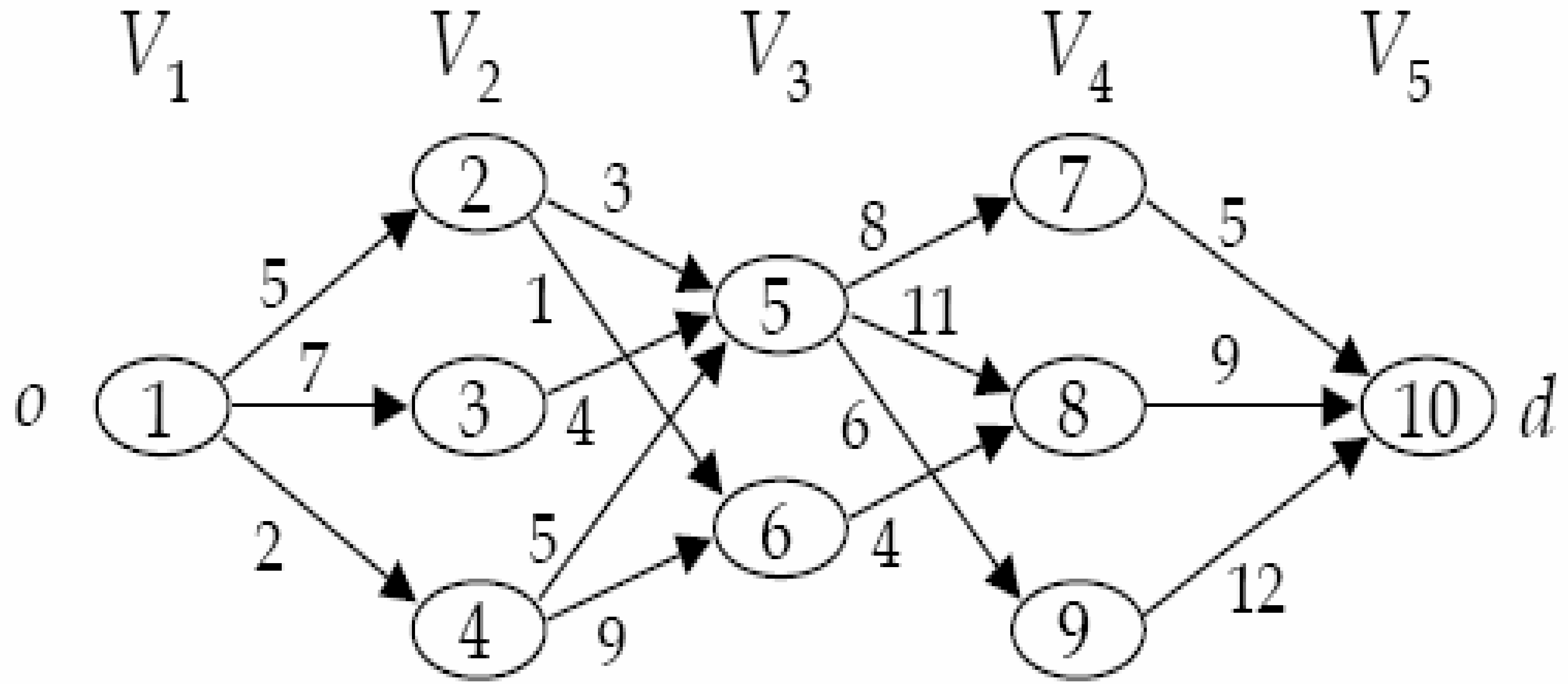
Aplicación

- **Problema de la mochila**
  - Si $\mathbf{O_1, O_2, ..., O_t}$, es una selección óptima de objetos para una mochila de capacidad **M** con objetos de un conjunto **C**
  - La subsolución es óptima al problema de una mochila con capacidad $\mathbf{M\text{-}P_t}$ ($\mathbf{P_t}$ = peso del objeto $\mathbf{O_t}$) y de conjunto de selección $\mathbf{C\text{-}\{O_t\}}$

Resolución de problema de optimización

- Ver si se puede aplicar el principio de optimalidad
- Localizar algoritmo en base el principio de optimalidad
- Localizar etapas en las que estructurar el proceso de resolución (“*resolución por etapas*”)
  - Dos subproblemas de una misma etapa son independientes
  - Un subproblema se estudia en una etapa en la que sus subproblemas ya se han resuelto en etapas anteriores
- Visión gráfica en forma de grafo
  - Los nodos son los subproblemas y las aristas determinan las dependencias entre subproblemas
  - Los nodos se agrupan en etapas

Método para PD



Backward - Fordward

Consideraciones

- Sea $Sol_{i,j}$ la solución a un nodo *j* (subproblema) en la etapa *i*
- Planteamiento *Backward*
  - $Sol_{i,j} = f(Sol_{i-1,j})$
- Planteamiento *Fordward*
  - $Sol_{i,j} = f(Sol_{i+1,j})$
- Terminología poco significativa
- Depende de la forma de "dibujar" el grafo

Método para PD

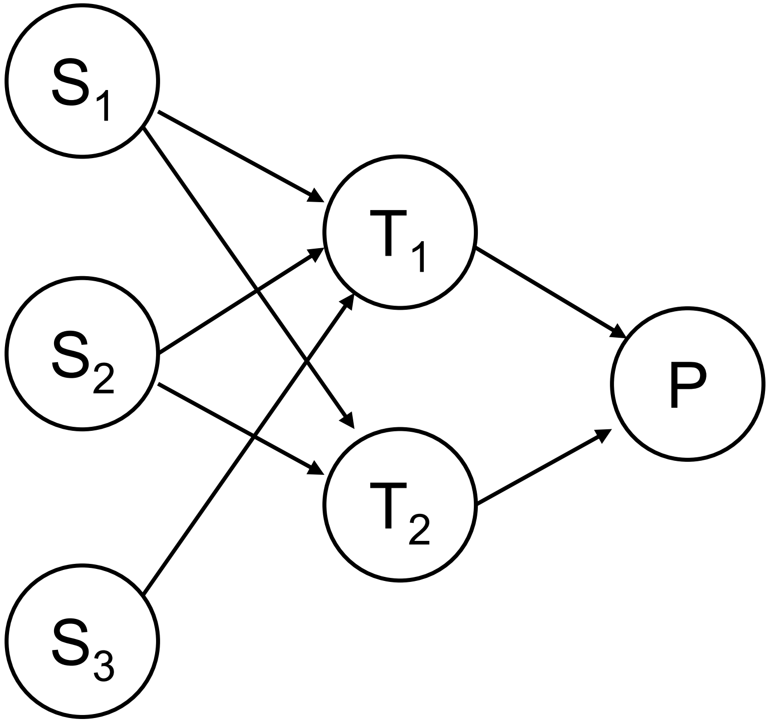
Consideraciones

- Proceso de división parecido a un DAC (combinando las soluciones para subproblemas más pequeños)
- Con el POB se dispone de mayor comodidad y flexibilidad en el tratamiento del problema
  - Generalmente algoritmos iterativos
  - Requerimiento de almacenamiento temporal para resultados asociados a subproblemas (utilización de tablas)
  - No es necesario calcular varias veces la solución para problemas pequeños

Iteración vs. recursión

- La solución de un subproblema puede ser necesitada en la resolución de más de un subproblema de otras etapas
- De forma iterativa, el subproblema se resuelve de **UNA SOLA VEZ** y su solución se almacena (tabla)
- De forma recursiva, el subproblema se resuelve **CADA VEZ** que se necesita
- Con versiones iterativas, aumenta considerablemente la eficiencia

Iteración vs. recursión



Resolución de P

- Resolución de T1
  - Res. S1, S2, S3
- Resolución de T2
  - Res. S1, S2, S3

Algoritmo Recursivo

Resolución de 9 subproblemas

Algoritmo Iterativo

Resolución de 6 subproblemas

Ejemplo

- Problema de la sucesión de Fibonacci con DAC

```
Fibonacci (n: integer)
    Si n<2 Devolver n
    Sino Devolver Fibonacci (n-1) + Fibonacci (n-2)
```

- Problemas
  - Cálculos repetidos
  - Complejidad exponencial
- Solución
  - Calcular los valores de menor a mayor empezando por 0, e ir guardando los resultados en una tabla

Ejemplo

- Problema de la sucesión de Fibonacci con PD

```
Fibonacci (n: integer)
    T[0]:= 0; T[1]:= 1;
    for i:= 2 to n do
        T[i]:= T[i-1] + T[i-2];
    Devolver T[n];
```

Ecuación Recurrente de Rendimiento Óptimo

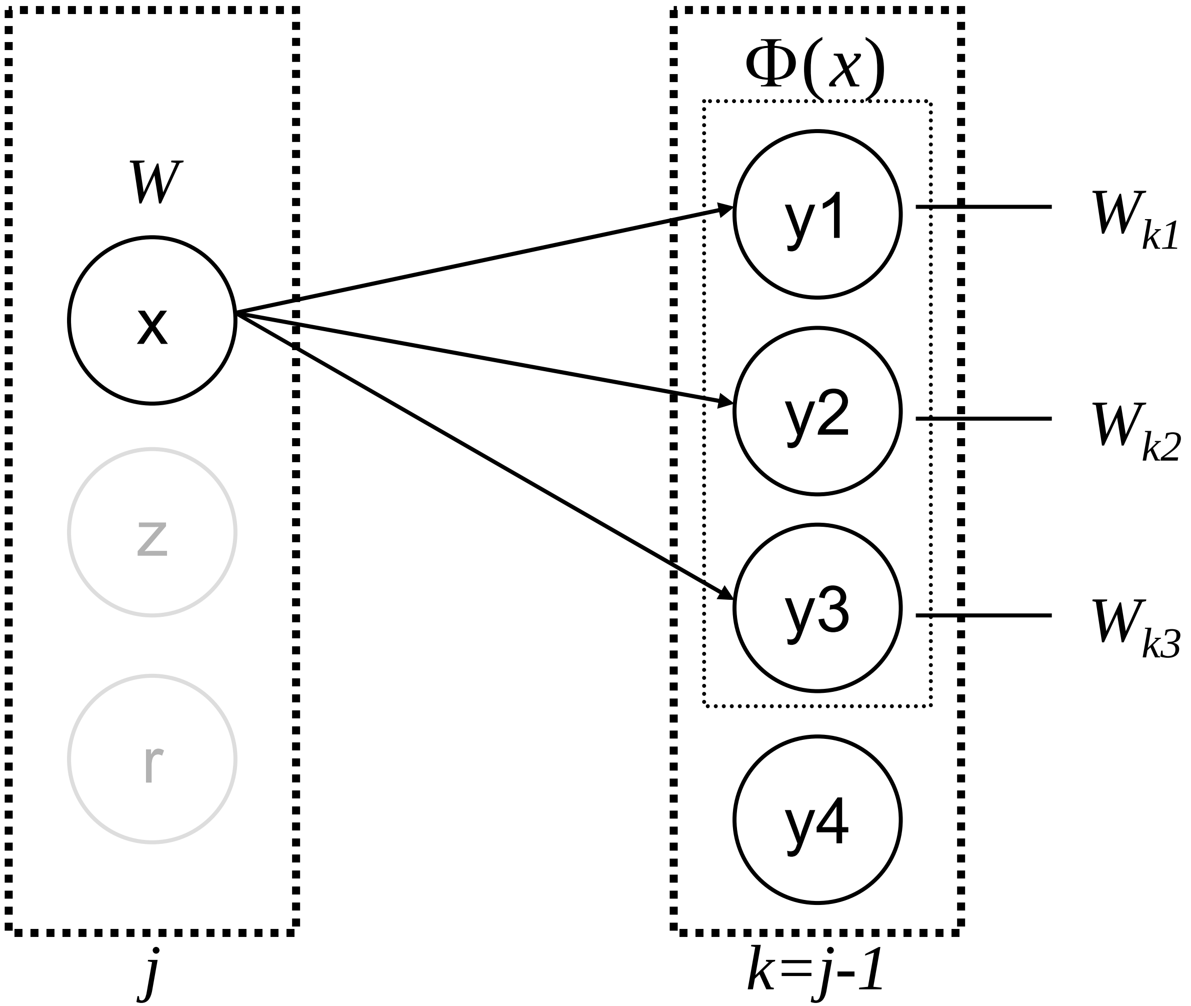
- Relación del coste (o valor) asociado a cada nodo (o subproblema) con otros nodos de otras etapas
- Forma general

$$W_j(x) = \underset{s}{óptimo}\, g\left(W_{k1}(y_1), W_{k2}(y_2), \ldots, W_{ks}(y_s)\right)$$

- Notación
  - $\boldsymbol{W_j}$: valor de *ERRO* para el nodo ***X*** en la etapa ***j***
  - ***S*** variación de subconjuntos del conjunto $\boldsymbol{\Phi(X)}$ de los nodos conectados a ***X***
  - ***g***: función de "*ganancia relativa*". Relaciona valores $\boldsymbol{W_k}$ de los nodos de ***S*** con los valores que puede tomar ***W*** en el nodo ***X***.

    De todos los posibles toma el óptimo (máximo o mínimo)

Notación



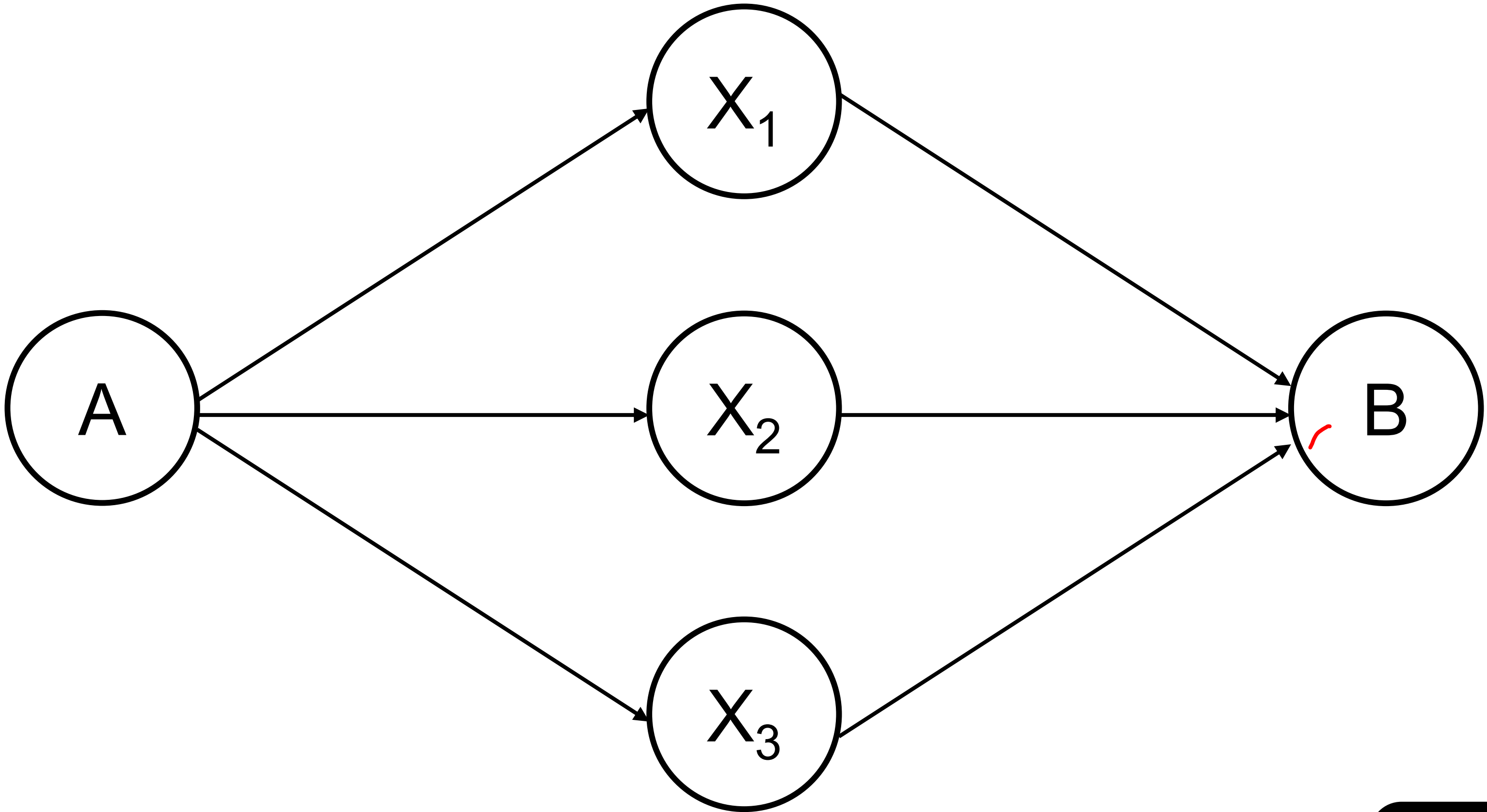
Diseño general del algoritmo de PD

- Planteamiento de la ecuación ERRO
- Refinamiento del siguiente esquema:

```
function DinamicoBW (conjunto_t S) {
        if (S==∅) return Solucion_Directa;
        else
                for (i=0; i<size(S); i++) {
                        x = elemento_i(S,i);
                        W[x] = DinamicoBW(S-{x}) + g(x,S);
                }
        return optimo(W);
}
```

Idea fundamental

- Dinámico (A,B) = mejor { coste(A,x_i)+Dinamico(x_i,B) }



Definición

- Grafo G(N,A) que tiene su vértices agrupados en una colección de *n* subconjuntos disjuntos V_1, V_2, …V_n llamados etapas del grafo cumpliendo:

  - GM1 $N = \bigcup_{i=1}^{n} V_i$
  - GM2 $\text{si } x, y \in V_i \Rightarrow (x, y), (y, x) \notin A$
  - GM3 $\text{si } (x, y) \in A, \text{con } x \in V_i, y \in V_j \Rightarrow i < j \ (\text{o } j = i + 1)$
  - GM4 $\text{Sea } \Phi_a(x) \text{ el conjunto de antecesores de un nodo x}$

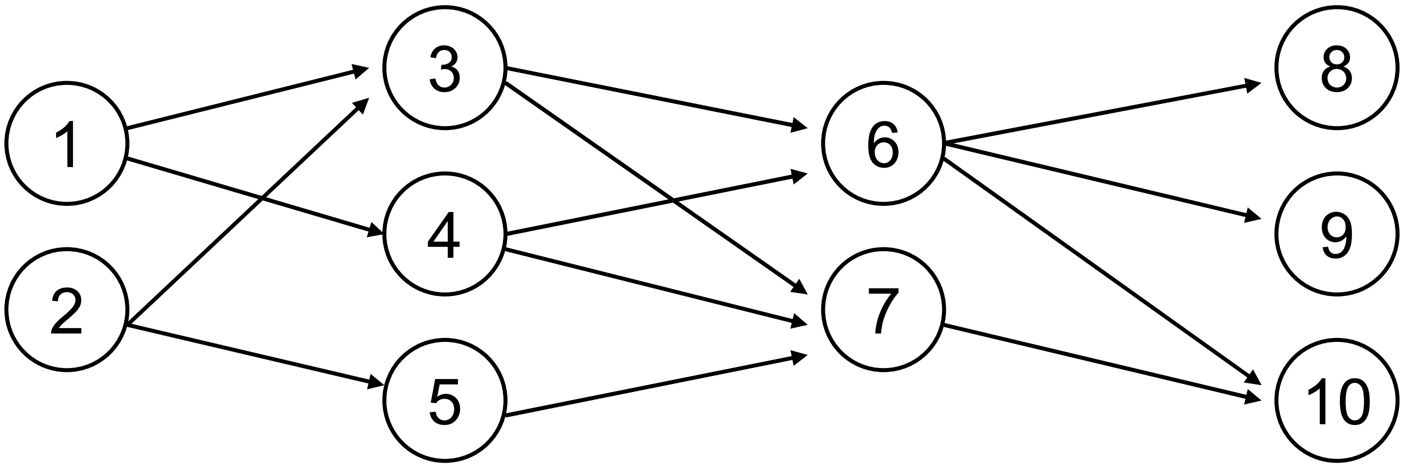
    $\text{Sea } \Phi_s(x) \text{ el conjunto de sucesores de un nodo x}$

$$\Phi_a(x) = \phi \text{ si } i = 1 \text{ y } \Phi_s(x) = \phi \text{ si } i = n$$

Definición

- En algunas ocasiones se impone una condición más:
  - GM5
    - $|V_1| = |V_n| = 1$

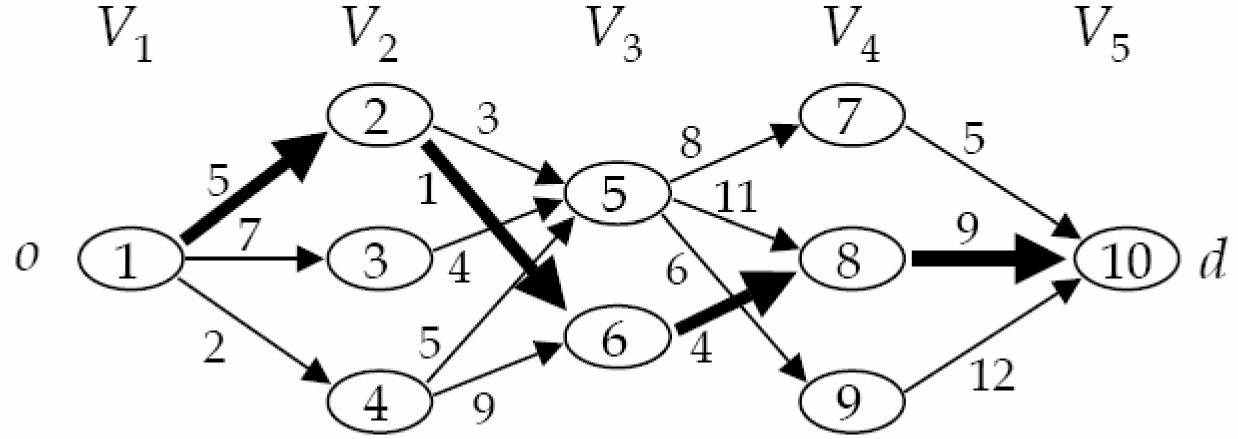
Ejemplo con n=4



Camino mínimo en un G.M.

Problema

- Grafo multietápico
- Arcos con un peso asociado
- Búsqueda del mejor camino para ir desde V_0 a V_n
- Aplicación del esquema de Programación Dinámica para buscar la solución



Aplicación del principio de optimalidad

- Suponemos como camino óptimo $\mu(x_{k1}, x_{k2}, ..., x_{kn})$
  - Si x_{ki}, $x_{kj} \in \mu$ el subcamino $m(x_{ki},\ldots,x_{kj})$ de μ es óptimo para el subproblema de ir desde x_{ki} a x_{kj}
  - Si $x_k \in \mu$, el subcamino $m(x_1,\ldots,x_k)$ es óptimo para el subproblema de ir desde V_1 hasta x_k, así como $m_2(x_k,\ldots,x_n)$ lo es para ir desde x_k hasta V_n
- Las dos afirmaciones se pueden demostrar por reducción al absurdo

Camino mínimo en un G.M.

Diseño del algoritmo

$\Phi_a(x)$ Conjunto de nodos antecesores de x

$f(x,y)$ Valor del arco (x,y)

- Numeración
  - Etapa 1 $\Rightarrow$ nodos de V_1
  - Etapa 2 $\Rightarrow$ nodos de V_2
- Pasos
  - Asignar los valores iniciales: $W_1(x) = 1 \ \forall x \in V_1$
  - Para i=2 hasta n, calcular los valores

$$W_i(y) = \underset{\substack{y \in \Phi_a(x) \\ j<y}}{óptimo}\left\{ f(x,y) + W_j(y) \right\}$$

  - Elegimos el valor y marcamos el nodo **y*** donde se alcance

$$WP = \underset{x \in V_n}{óptimo}\{W_n(x)\}$$

Algoritmo a partir de la ecuación

```
function cogm(vertice_t x, etapa_t e):vertice_t {
    if (e==1)
        return 0;
    else
        for (i=0; i<size(antecesores(x)); i++) {
                y = elemento_i(antecesores(x),i);
                m = etapa_nodo(y);
                w[x] = cogm(y,m) + peso(x,y);    // f(x,y)
        }
    return optimo(w);
}
```

Camino mínimo en un G.M.

Algoritmo iterativo

```
function cogc(vertice_t x):vertice_t {
    w[1] = 0;
    for (j=2; j<=x; j++) {
        r =elemento_optimo(antecesores(x));
        // r / w[r]+f(r,j) es optimo
        w[j] = w[r] + peso(r,j);
    }
    return w[x];
}
```

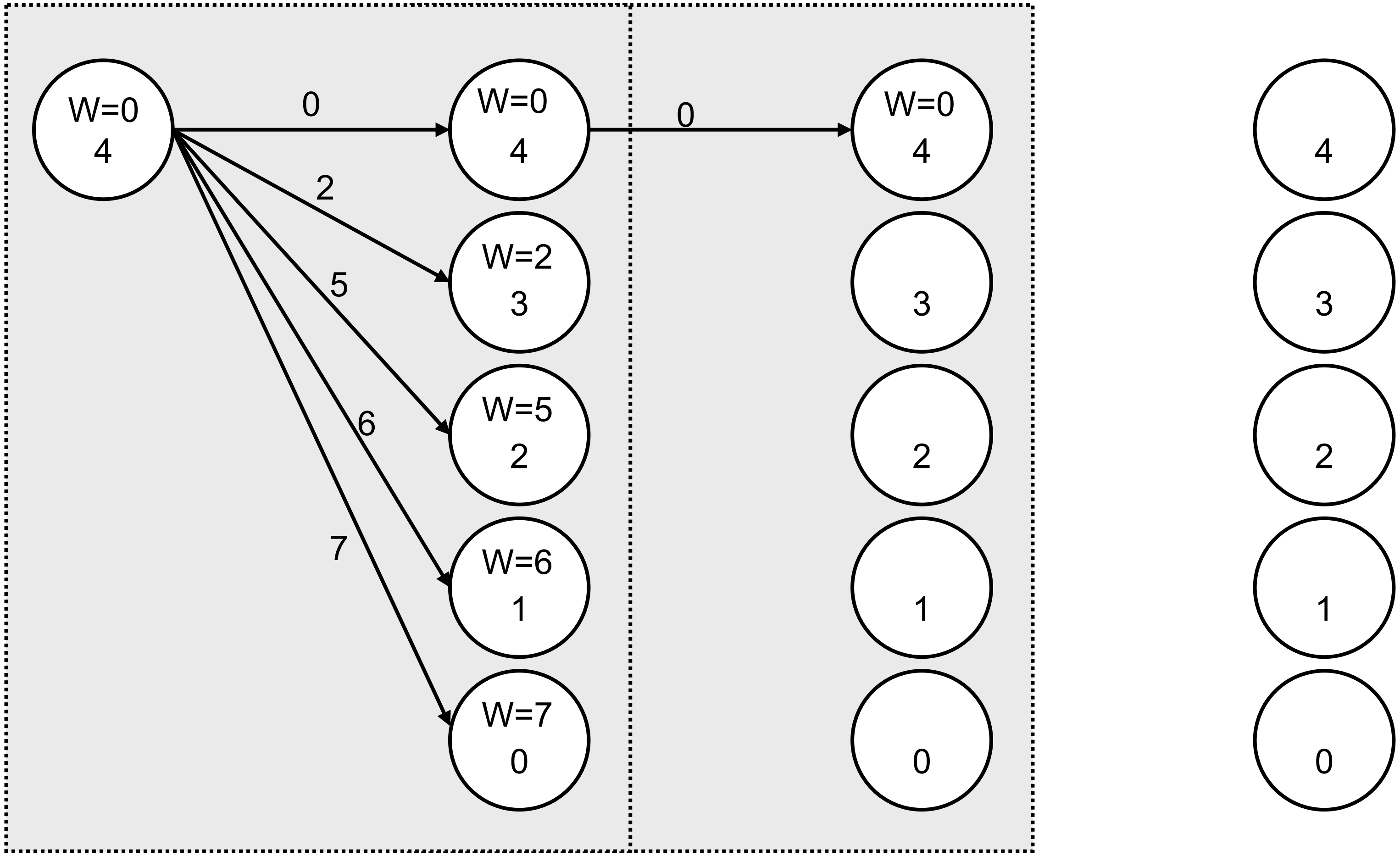
Aplicaciones

- Problemas para ser resueltos mediante P.D.
- Deben de ser “transformables” en un grafo multietápico
- A partir de él tenemos un algoritmo definido

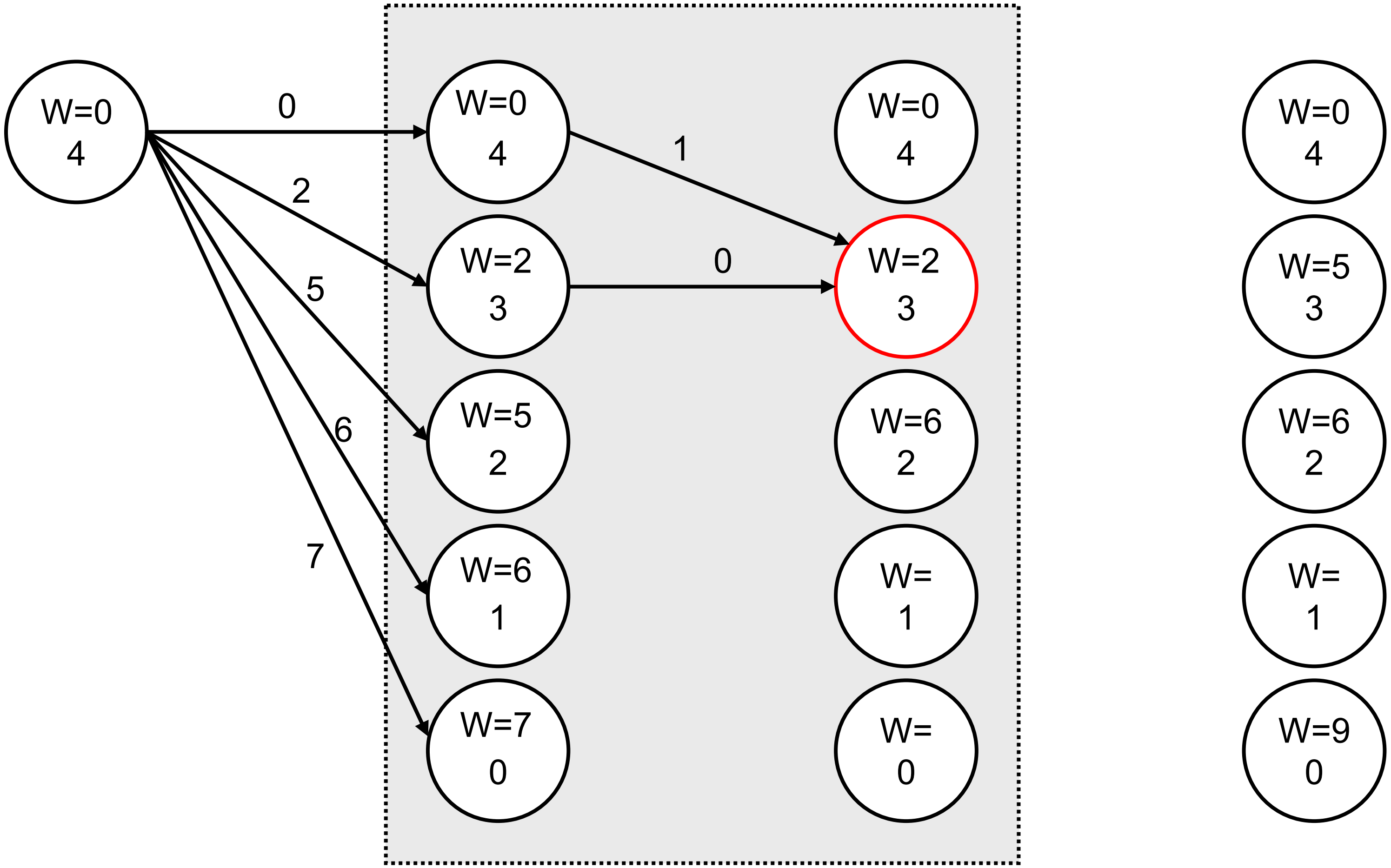
Ejemplo

- Problema de inversiones en entidades: *transformación*
  - Sistema que evoluciona a través de varios estados
  - Los estados se agrupan en etapas
  - En las etapas hay que tomar decisiones para pasar a otro estado de otra etapa
  - En el estado se representa el dinero disponible
  - Los pesos de los arcos es la cantidad que se invierte

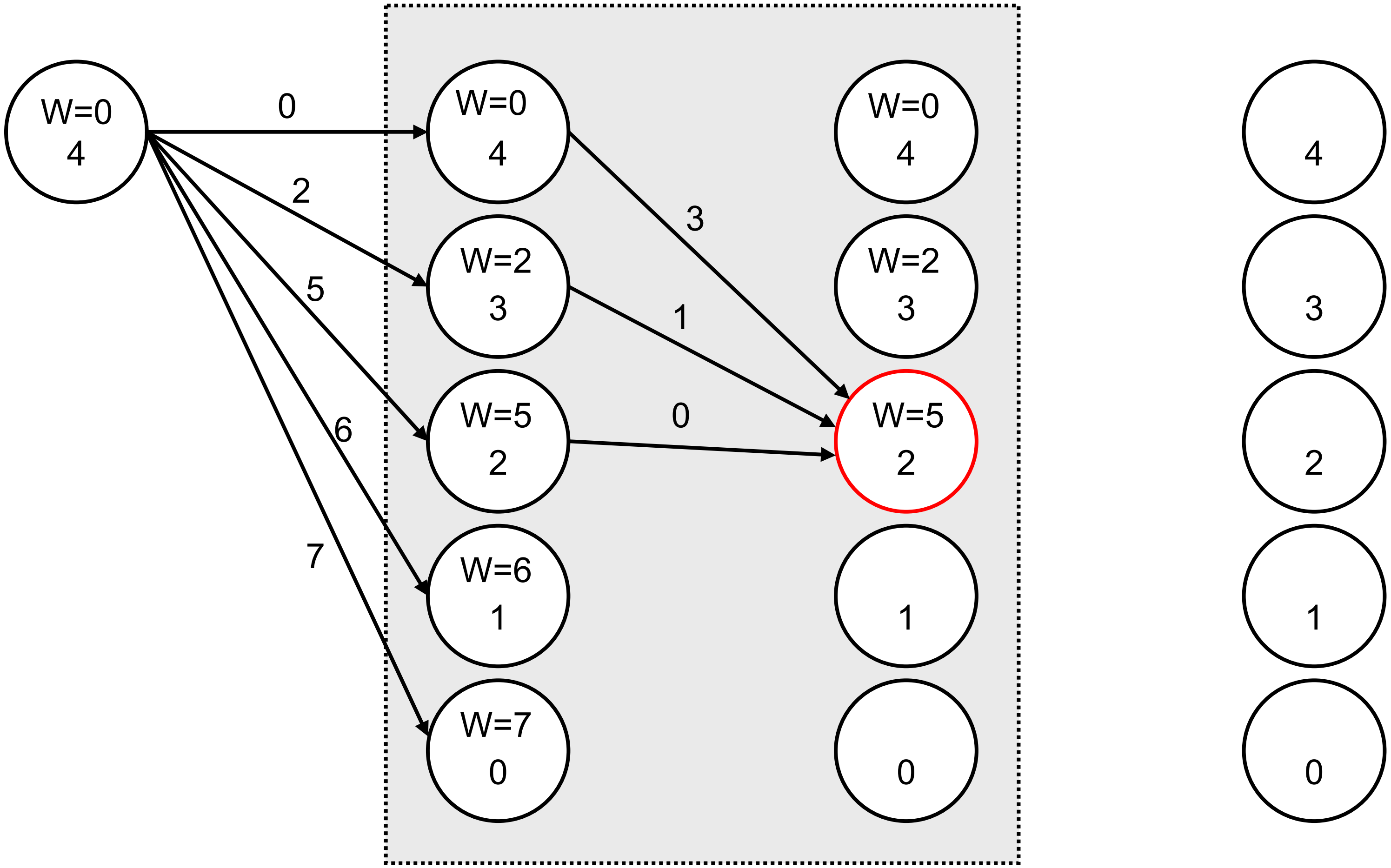
Problema de inversiones



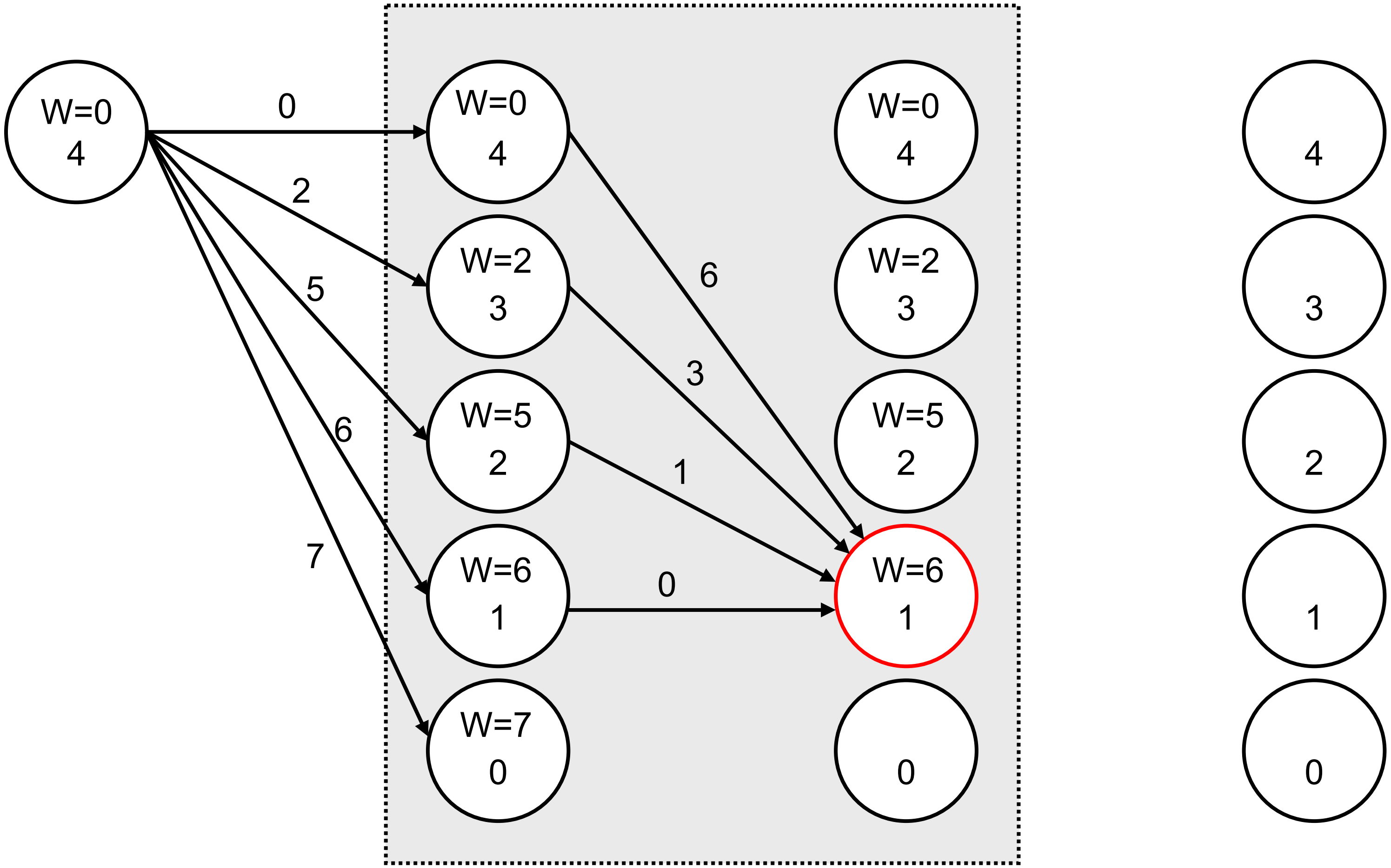
Problema de inversiones



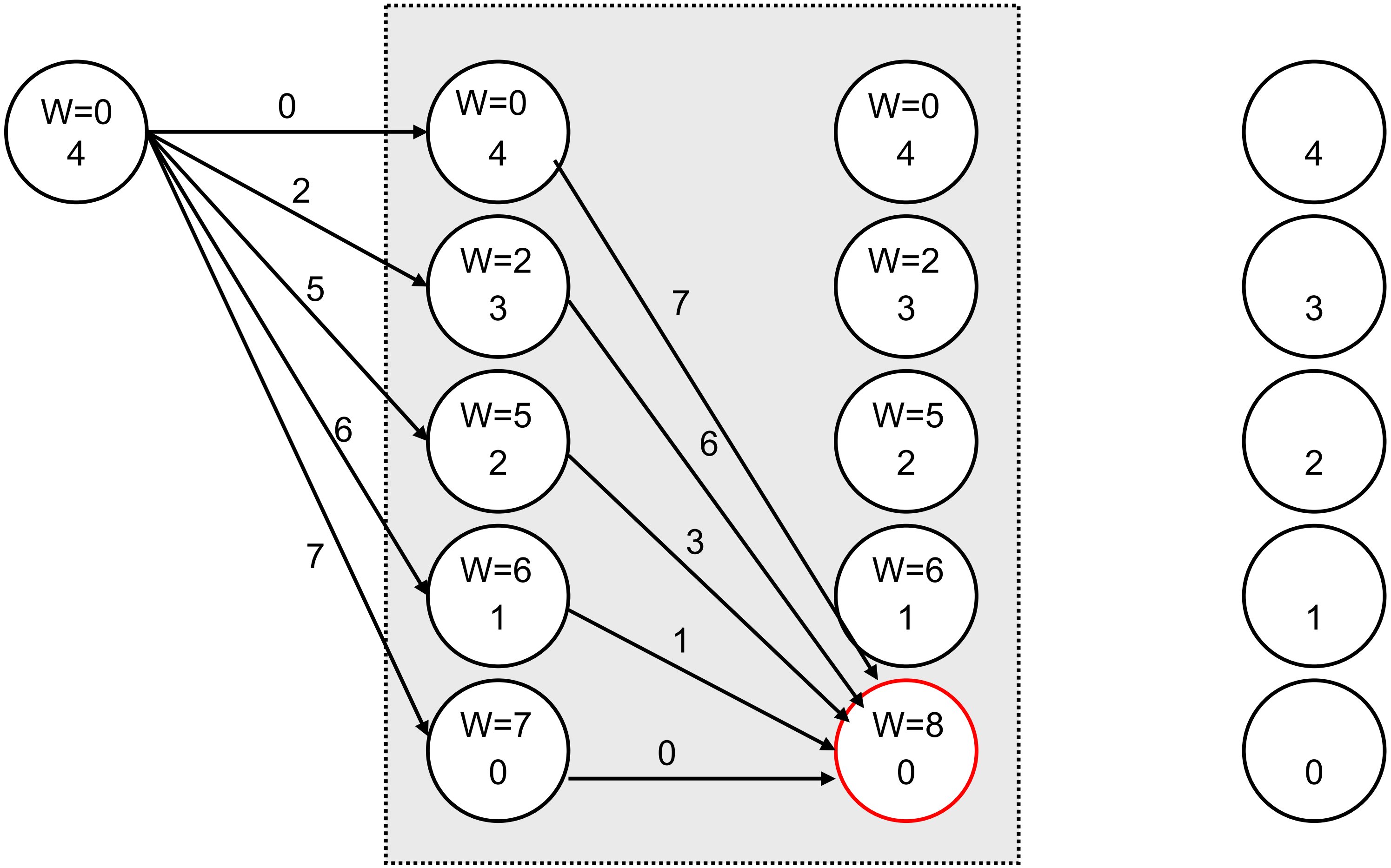
Problema de inversiones



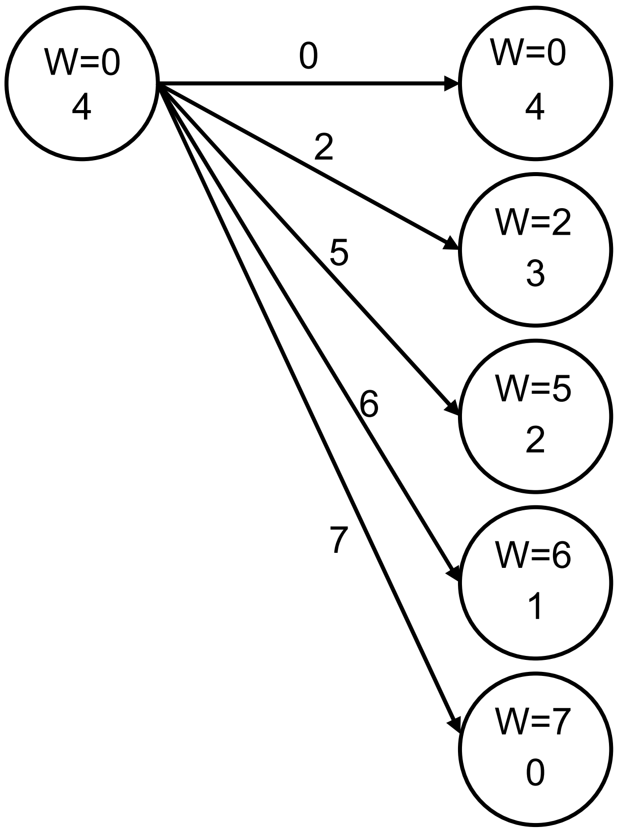
Problema de inversiones

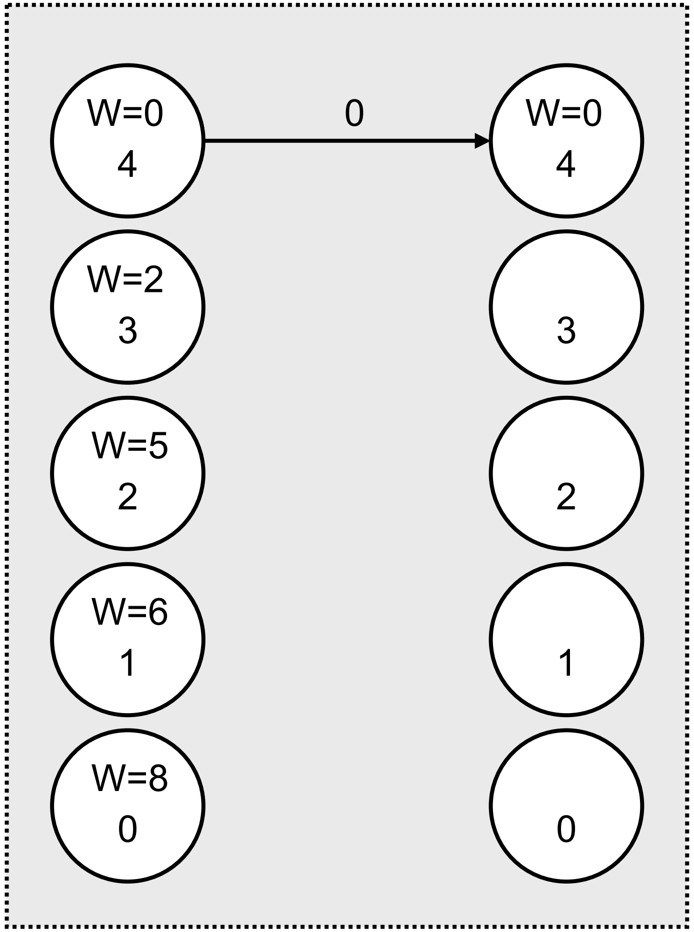


Problema de inversiones

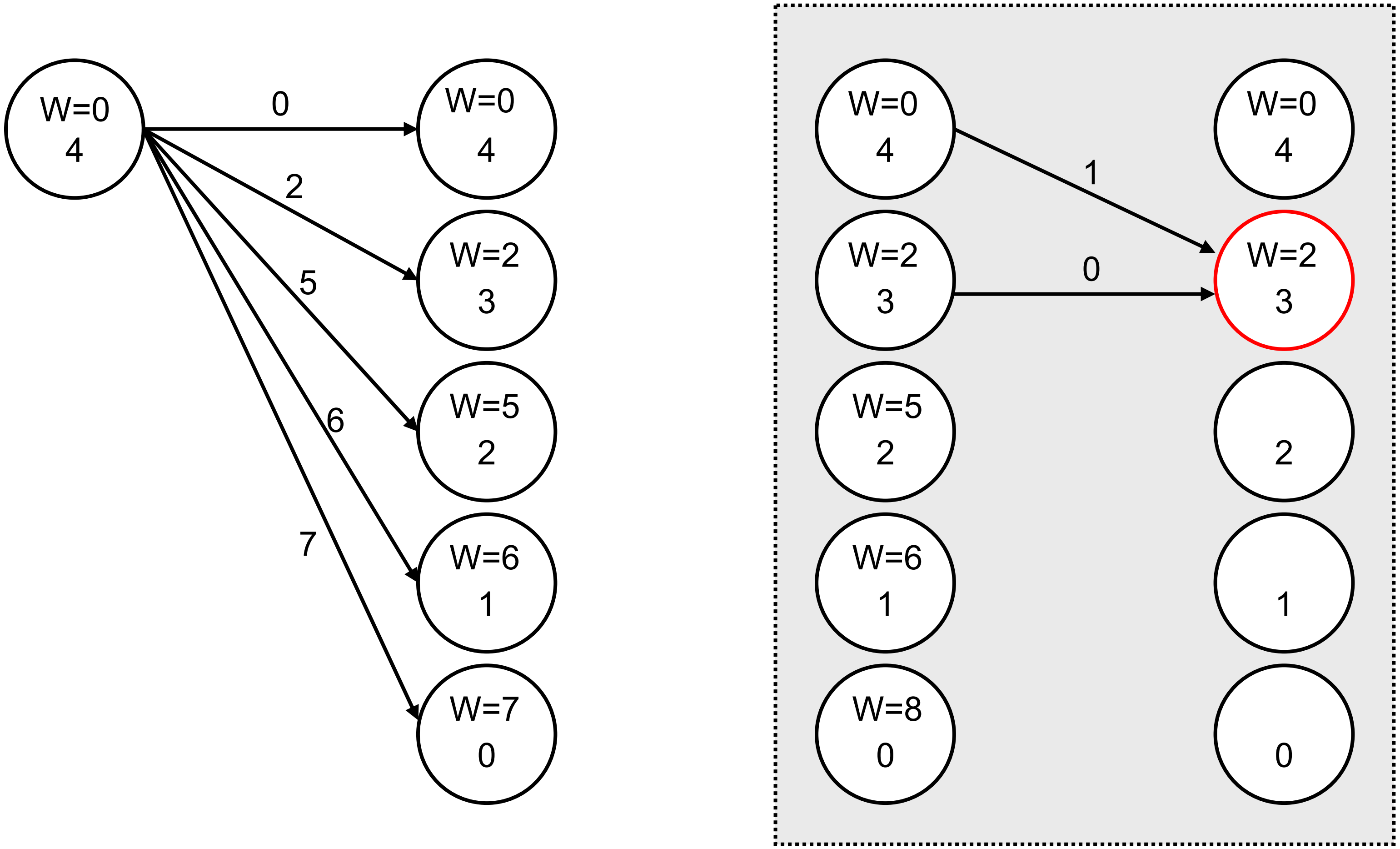


Problema de inversiones

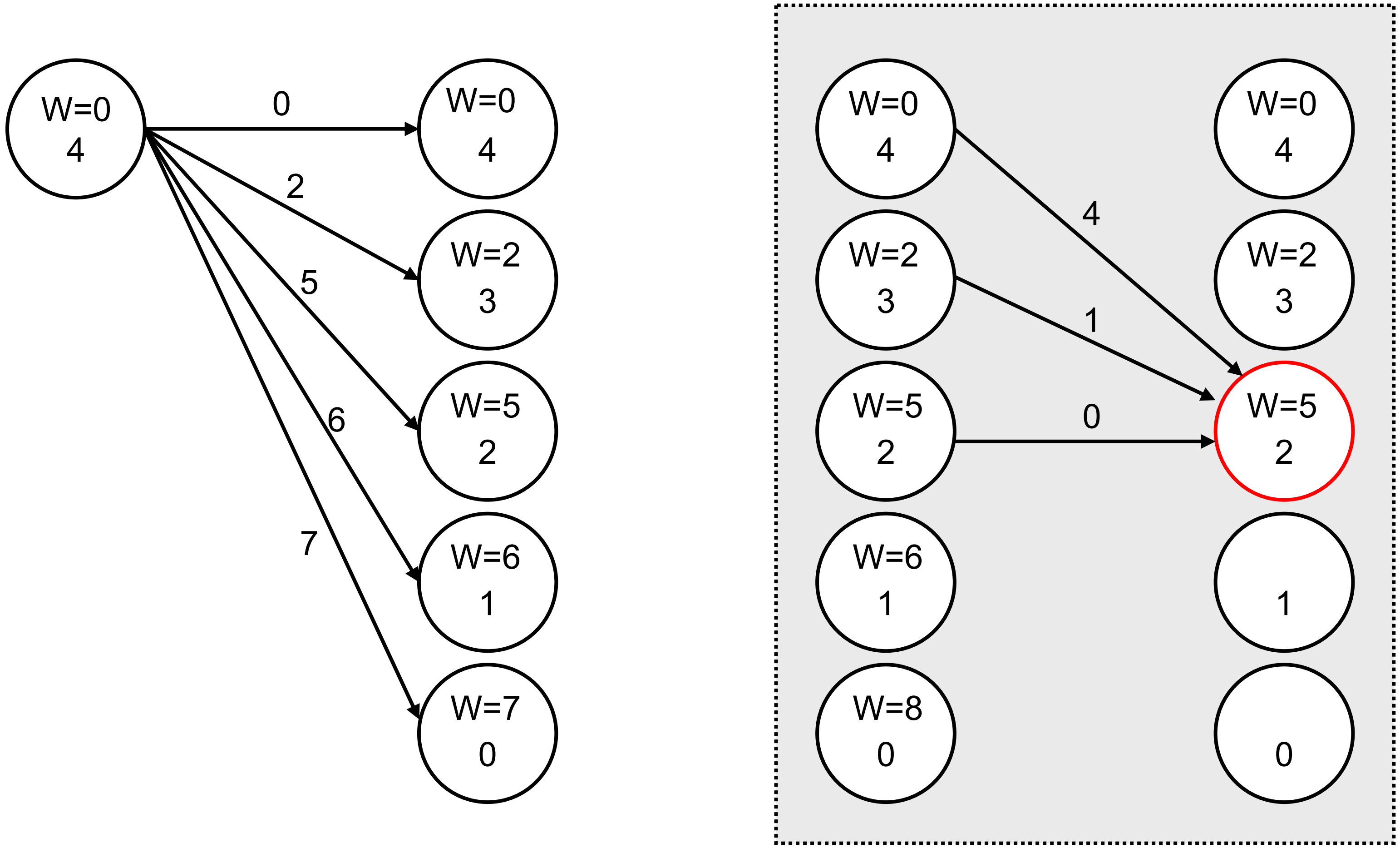




Problema de inversiones

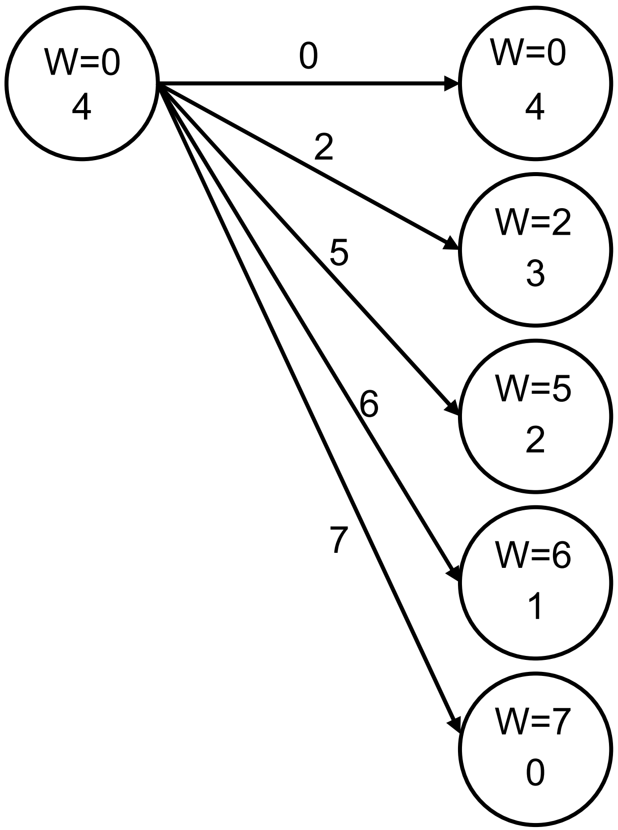


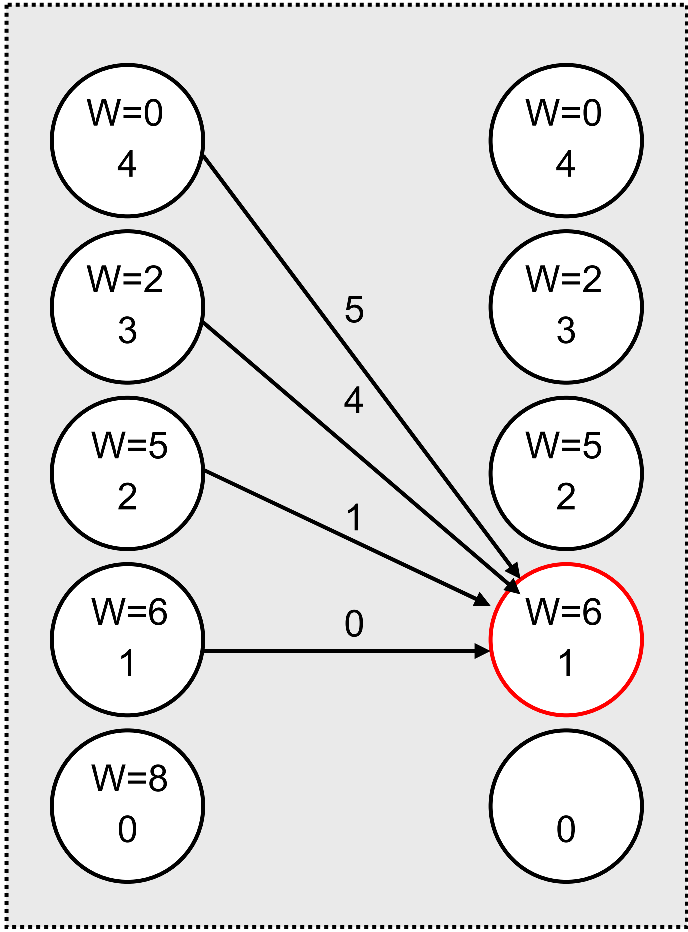
Problema de inversiones



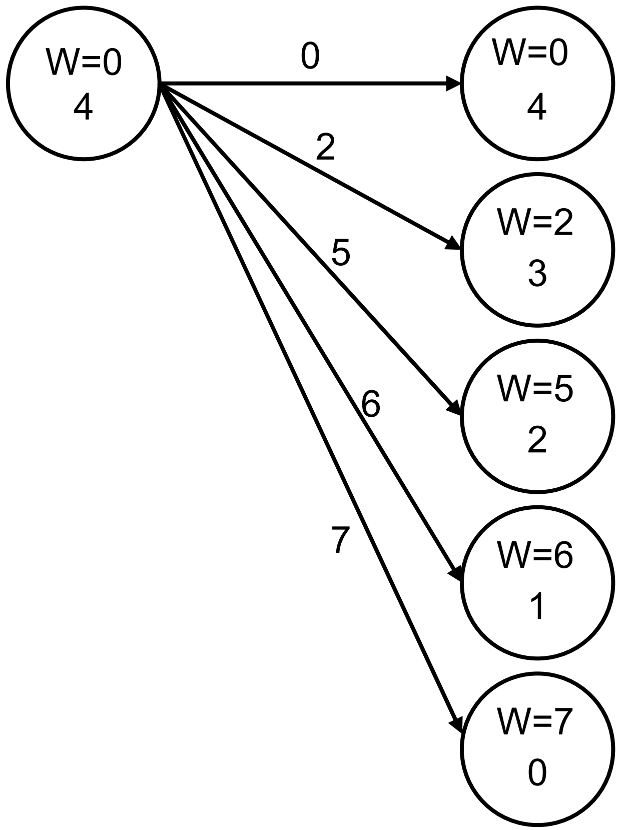
.:: Miguel Ángel Redondo ::.

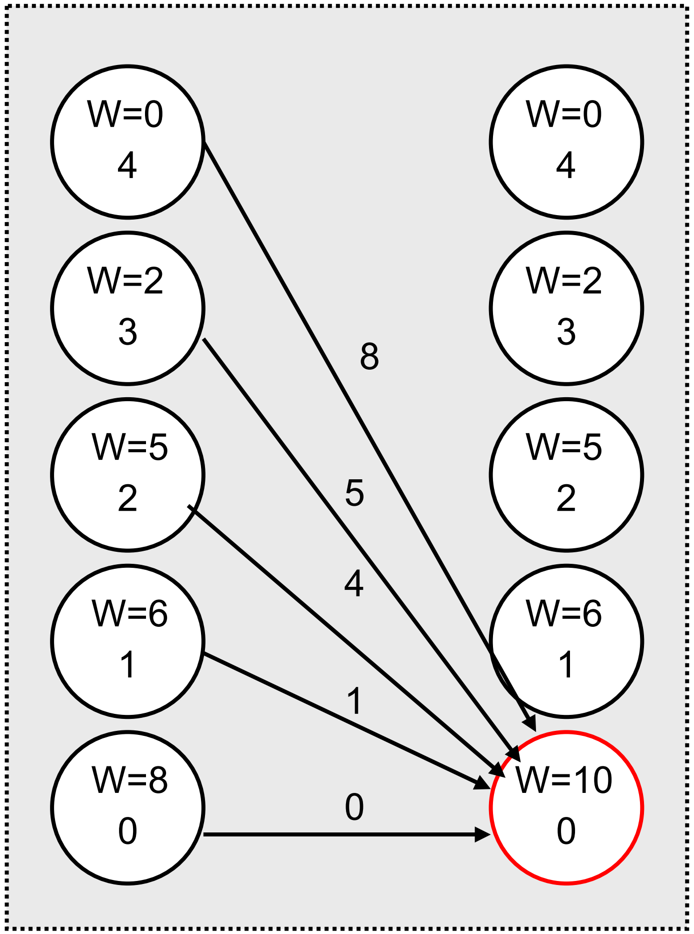
Problema de inversiones



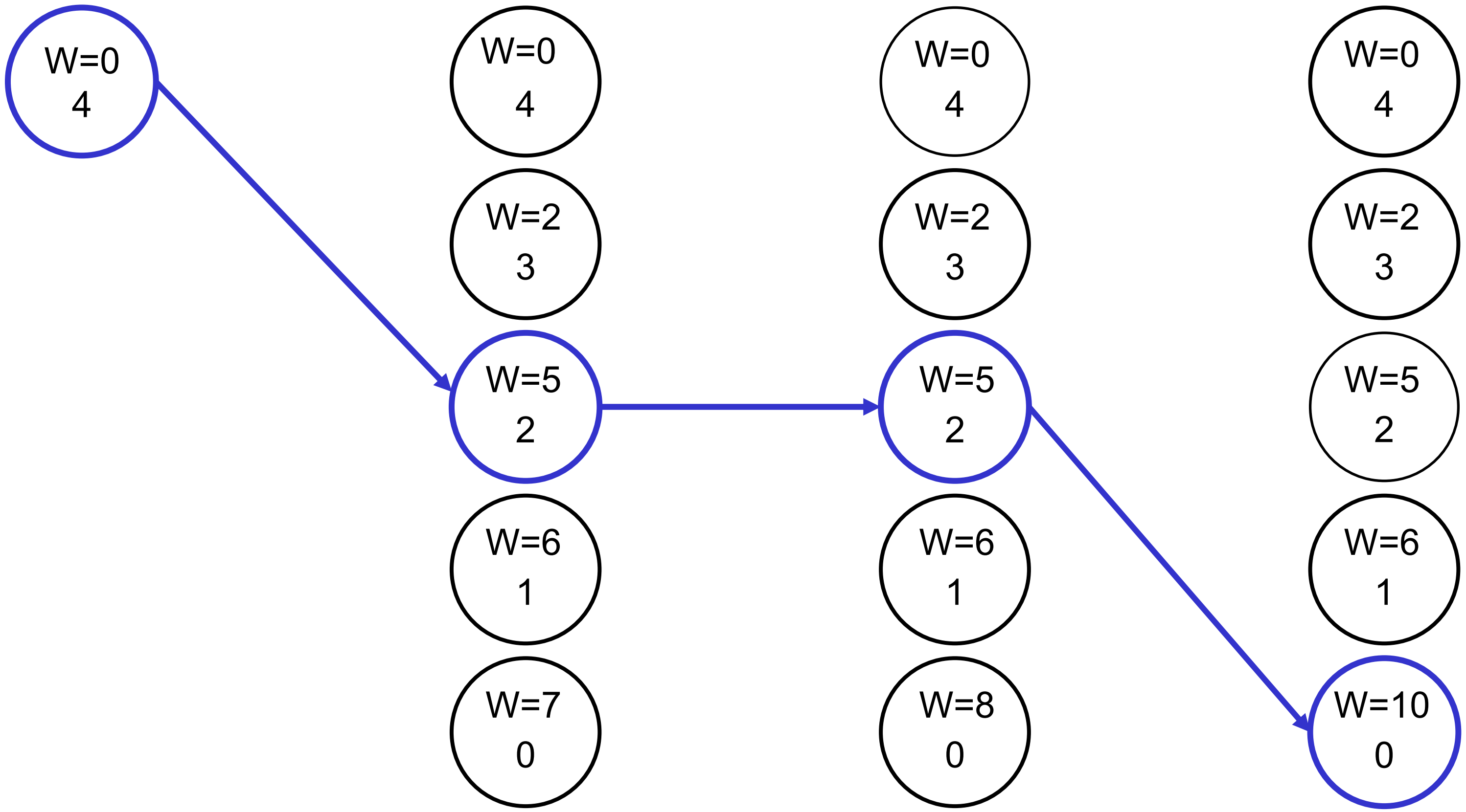


Problema de inversiones





Problema de inversiones



Camino mínimo entre vértices

Planteamiento

- Grafo G(N,A) dirigido y valorado
- Objetivo:
  - Buscar las distancias mínimas entre todo par de vértices del grafo
- Alternativas
  - Algoritmo de Dijkstra para cada nodo del grafo
  - Aplicación de técnicas de Programación Dinámica
    - Verificar Principio de Optimalidad de Bellman
    - Diseño del algoritmo

Aplicación del Principio de Optimalidad

- Se aplica el mismo razonamiento que para el problema del camino óptimo en un grafo multietápico
- Si *k* es un nodo intermedio en el camino más corto de *i* a *j*, entonces el fragmento de camino de *i* a *k*, y el fragmento de *k* a *j* son también óptimos

Diseño del algoritmo

- El camino sin ciclos entre dos vértices contiene un máximo de *n-1* arcos
- Un camino óptimo no puede tener ciclos
- Consideraremos *n-1* etapas
- En cada etapa *i* es estudiarán caminos de longitud *i*
- $W_i(x,y)$ (ERRO) quedará definida como la distancia mínima entre *x* e *y* en la etapa *i*
- El resultado de *W* en la última etapa es la solución al problema

Diseño del algoritmo

- Si C representa la matriz de adyacencia del grafo,

$$W_0(x, y) = C[x, y]$$

$$W_i(x, y) = \underset{i=1 \ldots n}{Min}\left\{W_{i-1}(x, y), W_{i-1}(x, i) + W_{i-1}(i, y)\right\}$$

```
function co(grafo_t c, int etapa, vertice_t x, vertice_t y) {
        if (n==0) return c[x,y];
        else return (
                minimo(co(c,n-1,x,y),
                        co(c,n-1,x,n-1) + co(c,n-1,n-1,y)));
}
```

Algoritmo de FLOYD

- Planteamiento anterior altamente ineficiente
- Mejoraría utilizando una matriz auxiliar D y con un enfoque iterativo

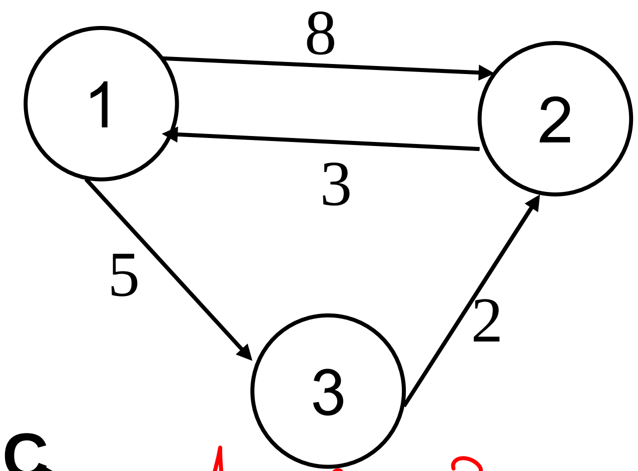
```
function floyd(grafo_t c): solucion_t D {
        D = c;
        for (i=1; i<=n; i++)
           for (j=1; i<=n; i++)
              for (k=1; k<=n; k++)
                   D[j,j] = minimo(D[j,k], D[j,i]+D[i,k]);
        return D;
}
```

$O(n^3)$

Dijkstra vs. FLOYD

- Mismo orden de complejidad
- Para un grafo con muchos arcos es mejor Floyd
- Para un grafo poco denso es más eficiente Dijkstra

Ejemplo de FLOYD



C

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

i=1 D1

0	8	5
3	0	8
∞	2	0

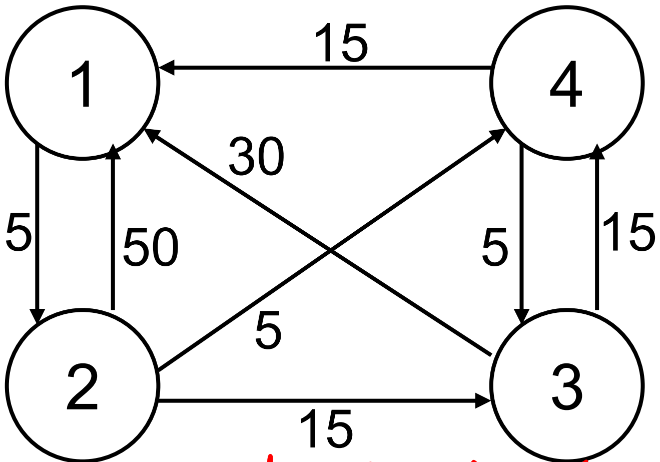
i=2 D2

0	8	5
3	0	8
5	2	0

i=3 D3

0	7	5
3	0	8
5	2	0

Ejemplo de FLOYD



$D_0 = C$

	1	2	3	4
1	0	5	∞	∞
2	50	0	15	5
3	30	∞	0	15
4	15	∞	5	0

i=1 D_1

0	5	∞	∞
50	0	15	5
30	35	0	15
15	20	5	0

i=2 D_2

0	5	20	10
50	0	15	5
30	35	0	15
15	20	5	0

i=3 D_3

0	5	20	10
45	0	15	5
30	35	0	15
15	20	5	0

i=4 D_4

0	5	15	10
20	0	10	5
30	35	0	15
15	20	5	0

Ejercicio

Una compañía de ferrocarriles da servicio a ***n*** estaciones ***S1,...,Sn*** y trata de mejorar su servicio al cliente mediante terminales de información.

Dadas una estación origen ***So*** y una estación destino ***Sd***, un terminal debe ofrecer (inmediatamente) la información sobre el horario de los trenes que hacen la conexión entre ***So*** y ***Sd*** y que minimizan el tiempo de trayecto total.

Se pide implementar un algoritmo que realice esta tarea a partir de la tabla con los horarios, suponiendo que las horas de salida de los trenes coinciden con las de sus llegadas (es decir, que no hay tiempos de espera) y que, naturalmente, no todas las estaciones están conectadas entre sí por líneas directas; así, en muchos casos hay que hacer transbordos aunque se supone que tardan tiempo cero en efectuarse.

SUGERENCIAS Y REFLEXIONES

Sea ***T(i,j,V)*** el tiempo del trayecto mínimo para ir de la estación de origen ***i*** a la estación destino ***j***, pudiendo utilizar como estaciones intermedias las contenidas en el conjunto ***V***. Denominar ***L(i,j)*** al tiempo del trayecto directo de ***i*** a ***j***, siendo ∞ si esta conexión no existe.

La idea general es la de comprobar si es más beneficioso ir de forma directa o a través de cada uno de los posibles caminos. Esto hay que comprobarlo para cada par ***(i,j)***.

¿Pude representarse el problema mediante un grafo siendo las estaciones los vértices del grafo y las aristas las conexiones entre dos estaciones?

Planteamiento

- Mochila de capacidad ***M***
- Maximizar $\sum_{i=1}^{n} b_i x_i$
- Sujeto a $\sum_{i=1}^{n} p_i x_i \leq M$
- Con $x_i \in \{0,1\}, 0 \leq i \leq n$
- Algoritmo voraz no siempre encuentra solución óptima

Principio Optimalidad de Bellman

- Si $S_1 = \{O_1, O_2, ..., O_{t-1}, O_t\}$ es solución óptima para una mochila M
- Entonces $S_2 = \{O_1, O_2, ..., O_{t-1}\}$ es óptima para una mochila de capacidad M-P_t con un conjunto de candidatos C-{O_t}
- P_t es el peso del objeto O_t

Notación

- X_i=1 si el objeto i está en la mochila
- $\sum_{i=1}^{t} x_i b_i$ es óptimo con $\sum_{i=1}^{t} x_i p_i \leq M$

Principio Optimalidad - demostración

- Solución óptima $\{x_1, x_2, ..., x_{t-1}, x_t\}$
- Probamos
  - Eliminando el último objeto la subsolución es óptima
  - Reducción al absurdo
- Hipótesis

$$X = \{x_1, ..., x_{t-1}, x_t\}$$

$$\sum_{i=1}^{t} x_i b_i \text{ es óptimo con } \sum_{i=1}^{t} x_i p_i = M$$

Principio Optimalidad - demostración

- Suponemos $\{x_1, x_2, ..., x_{t-1}\}$ no es óptima
- Entonces existe $\{y_1, y_2, ..., y_{t-1}\}$ que cumple

$$\sum_{i=1}^{t-1} y_i b_i > \sum_{i=1}^{t-1} x_i b_i \text{ con } \sum_{i=1}^{t-1} y_i p_i = M - x_t p_t$$

- Sumamos a los dos términos $x_t b_t$

$$\sum_{i=1}^{t-1} y_i b_i + x_t b_t > \sum_{i=1}^{t} x_i b_i$$

- Esta solución es mejor de X lo que contradice la hipótesis de que X sea óptima

Diseño del algoritmo

- En cada etapa se estudiará cada uno de los objetos del problema
- Planteamiento backward
- L… timo … d… ncia … a…
- Ecuación recurrent…

Inclusión del objeto n
Quedan n-1 objetos
Ganamos b_n
Queda capacidad $M-P_n$

Desechamos el objeto n
No ganamos nada
Queda capacidad M

$$W_n(x) = Max\left\{W_{n-1}(M - p_n) + b_n, W_{n-1}(M)\right\}$$

Diseño del algoritmo

- Generalización de la ecuación recurrente

$$W_1(x) = 0$$

$$W_i(x) = Max\{W_{i-1}(x - p_i) + b_i, W_{i-1}(x)\}$$

- Eficiencia versión recursiva

$$T(n) \in O(2^n)$$

- Eficiencia versión iterativa

$$T(n) \in O(2^n) \longrightarrow T(n) \in O(2^{n/2})$$

Código del algoritmo

```
int mochila (int M, sol_t x, conjunto_t objs) {
        if (n==1) return(0);
        else {
                m1 = mochila(m,x,n-1);
                x1=x;
                m2 = mochila(m-objs[n].p,x,n-1)+objs[n].b;
                x2=x;
                if (m1>m2) {
                        x=x1;    x[n] = 0;          return(m1);
                } else {
                        x=x2;    x[n] = 1;          return(m2);
                }
        }
}
```

Otro planteamiento

- Grafo multietápico
- Similitud con problema de inversiones
- Problema propuesto cuya entrega y presentación en clase será valorada (*siempre de forma positiva*)

Problema

- Encontrar un recorrido de longitud mínima para un viajante que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades

- Dado un grafo dirigido con arcos de longitud no negativa, se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes (*circuito hamiltoniano*)

Notación

- ***G=(V,A)*** un grafo orientado
- ***V****={1,2,…,n}*,
- L_{ij} la longitud de $(i,j) \in A$
- $L_{ij}=\infty$ si no existe el arco ***(i,j)***.
- El circuito buscado
  - empieza en el vértice 1.
  - se compone de ***(1,j)***, con j≠1, seguido de un camino de ***j*** a 1 que pasa exactamente una vez por cada vértice de ***V \ {1,j}***
- **Principio de optimalidad**
  - si el circuito es óptimo, el camino de ***j*** a ***1*** debe serlo también

Ecuación de Rendimiento Óptimo

- Sea
  - ***S⊆V\{1}*** un subconjunto de vértices
  - ***i∈V\S*** un vértice

- Llamamos
  - ***g(i,S)*** a la longitud del camino mínimo desde ***i*** hasta ***1*** que pase exactamente una vez por cada vértice de ***S***

- Entonces:

$$long\_co = g(1, V \setminus \{1\}) = \min_{2 \le j \le n} \left\{ L_{ij} + g(j, V \setminus \{i, j\}) \right\}$$

Solución general

- Considerando ***i≠1***, ***S≠Ø*** e ***i∉S***:

$$g(i,S) = \min_{j \in S}\left\{L_{ij} + g(j, S \setminus \{j\})\right\} \qquad (1)$$

$$g(i,\phi) = L_{i1} \qquad i = 2,3,\ldots,n$$

- Método de resolución
  - Usar (1) y calcular ***g*** para todos los conjuntos ***S*** con un solo vértice (distinto del 1)
  - Volver a usar (1) y calcular ***g*** para todos los conjuntos S de dos vértices (distintos del 1) y así sucesivamente
  - Cuando se conoce el valor de ***g*** para todos los conjuntos S a los que sólo les falta un vértice (distinto del 1) basta calcular ***g(1,V\{1})***

Ejemplo

- Grafo con cuatro vértices definido por L

$$L = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

- Inicialización

g(2,∅)=5 g(3,∅)=6 g(4,∅)

Ejemplo

- Aplicamos la expresión (1) considerando conjuntos de un solo elemento
  - $g(2,\{3\}) = L_{23} + g(3,\varnothing) = 9 + 6 = 15$
  - $g(2,\{4\}) = L_{24} + g(4,\varnothing) = 10 + 8 = 18$
  - $g(3,\{2\}) = L_{32} + g(2,\varnothing) = 13 + 5 = 18$
  - $g(3,\{4\}) = L_{34} + g(4,\varnothing) = 12 + 8 = 20$
  - $g(4,\{2\}) = L_{42} + g(2,\varnothing) = 8 + 5 = 13$
  - $g(4,\{3\}) = L_{43} + g(3,\varnothing) = 9 + 6 = 15$

Ejemplo

- Aplicamos la expresión (1) considerando conjuntos de dos elementos
  - $g(2,\{3,4\}) = \min \{ L_{23} + g(3,\{4\}), L_{24} + g(4,\{3\}) \} =$

    $= \min \{29,25\} = 25$
  - $g(3,\{2,4\}) = \min \{ L_{32} + g(2,\{4\}), L_{34} + g(4,\{2\}) \} =$

    $= \min \{31,25\} = 25$
  - $g(4,\{2,3\}) = \min \{ L_{42} + g(2,\{3\}), L_{43} + g(3,\{2\}) \} =$

    $= \min \{23,27\} = 23$

Ejemplo

- Finalmente
  - $g(1,\{2,3,4\}) = \min \{ L_{12} + g(2,\{3,4\}),$
    $L_{13} + g(3,\{2,4\}),$
    $L_{14} + g(4,\{2,3\}) \} =$
    $= \min \{ 35, 40, 43 \} = 35$

Ejemplo

- Para saber/etiquetar/identificar el recorrido óptimo
  - Utilizamos una función adicional
  - ***J(i,S)*** es el valor ***j∈S*** que minimiza ***g(i,S)*** al aplicar la expresión **(1)**
- Aplicación en el ejemplo
  - J(1,{2,3,4}) = 2
  - J(2,{3,4}) = 4
  - J(4,{3}) = 3
  - 1

Implementación recursiva

```
int g(grafo_t L, int i,conjunto_t S) {
        int másCorto, distancia, j;
        if (conjunto_vacio(S)) {
                return L[i,1];
        else {
                masCorto = MAXINT;
                for (j=0; j<tamanyo(S); j++) {
                        distancia = L[i,j] + g(L,j,elimina(S,j));
                        if (distancia < masCorto)
                                masCorto = distancia;
                }
        }
        return másCorto;
}
```

Implementación recursiva

```
int g(grafo_t L, int i,conjunto_t S) {
        int másCorto, distancia, j;
        if (conjunto_vacio(S)) return L[i,1]
        else {
                if (gtab[i,S] ≥ 0) return gtab[i,
                else {
                        MasCorto = MAXINT;
                        for (j=0; j<tamanyo(S); j++) {
                                distancia = L[i,j] + g(j,elimina(S,j));
                                if (distancia < masCorto)
                                        masCorto = distancia;
                        }
                }
        }
}
```

Se utiliza una "***función con memoria***: *gtab*"
La tabla *gtab* contiene elementos que se inicializan a -1

Aumento de la eficiencia

- Con la programación dinámica
  - Es posible que se resuelvan subproblemas irrelevantes
  - Se saben que son irrelevantes cuando posteriormente se determina que no se necesitan
  - Puede haber subproblemas que se resuelven varias veces
- Enfoque recursivo
  - Sencillez en el diseño
  - Imposibilidad para establecer restricciones que resuelvan lo anterior
- Enfoque iterativo
  - Dificultad en el diseño
  - Facilidad para imponer restricciones

Utilización de una función con memoria

- Se añade una tabla al programa al programa recursivo con información de las soluciones de los subproblamas que ya se han resuelto
- Ante una llamada recursiva, primero se examina la tabla para ver si el subproblema ya ha sido resuelto
  - Si ya se ha resuelto se recupera la solución de la tabla
  - Si todavía no se ha resuelto se realiza la llamada recursiva

Problema Procesadores

N trabajos han de procesarse en un sistema que cuenta con **dos procesadores**. Para cada trabajo se saben los tiempos $\boldsymbol{a_i}$ y $\boldsymbol{b_i}$ que necesitaría el proceso en cada procesador.

Diseñar un algoritmo con *Programación Dinámica* que encuentre la solución óptima, esto es, la que minimiza el tiempo necesario para concluir todos los procesos.

Problema Procesadores

- En la etapa ***k*** se decidirá que procesador realizará el trabajo ***k***, esta decisión se hará en base al tiempo total de ejecución que se consumirá considerando las dos posibilidades (procesadores)

- Ecuación recurrente
  - Procesar(T,1) =

    Mejor{Ejecutar 1 con Proc1), Ejecutar 1 con Proc2}
  - Procesar(T,K) =
    - Mejor{ (Ejecutar k con Proc1)+Procesar(T,k-1),

      (Ejecutar k con Proc2)+Procesar(T,k-1) }

Problema Procesadores

Estructuras de datos a utilizar

- Para los trabajos
  - Vector bidimensional de ***n*** posiciones en una dimensión y dos posiciones en la otra posición. Contendrá los tiempo que tardan cada procesador en realizar cada trabajo
- Para la solución
  - Vector de ***n*** posiciones. En cada posición se indicará que procesador realiza el trabajo ***i***

Problema Diana

Diseñar un algoritmo que utilice Programación Dinámica para que determine cómo sumar exactamente **100** puntos con el menor número de dardos