Estructuras de Datos
Clase 17 – Grafos (Tercera Parte)

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Algoritmos para encontrar caminos en digrafos pesados con números reales

- <u>DFS pinchado (st-path search):</u> Permite encontrar un camino entre dos vértices s y t
- BFS para hallar camino con cantidad mínima de arcos.
- <u>DFS con backtracking, marca y desmarca:</u> Permite encontrar un camino de costo mínimo entre dos vértices s y t
- <u>Dijkstra:</u> Permite hallar todos caminos de costo mínimo entre un vértice *a* y todos los otros vértices
- <u>Floyd:</u> Permite hallar el camino de costo mínimo entre cada par de vértices s y t.

DFS pinchado (s-t path search)

Permite hallar un camino en el digrafo G entre Origen y Destino y retorna el camino hallado en Camino (el cual es una lista vacía al principio).

Si encontró un camino, retorna verdadero, en caso contrario retorna falso.

<u>**Algoritmo**</u> HallarCamino(G, origen, destino, camino) : boolean

origen.put(Estado, Visitado)

camino.addLast(origen)

Si origen = destino **entonces**

    retornar verdadero

Sino

    para cada adyacente v de origen en G **hacer**

        si v.get(Estado) = NoVisitado **entonces**

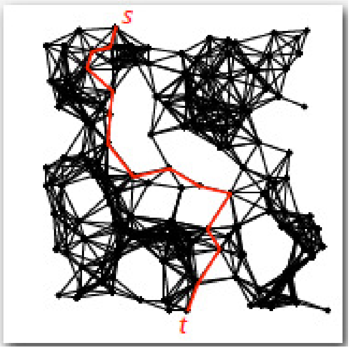
            encontre ← HallarCamino(G, v, destino, camino)

            si encontre **entonces retornar** verdadero

{ Cuando no encontré, hago backtracking: Borro origen del camino. }

Camino.remove(camino.last())

retornar falso



Complejidad:
T(n,m) = O(n+m)

Ejercicio: Codificar en Java

Estrategia para hallar ciclos

{ Encuentra un ciclo que contenga a v, buscando caminos desde los adyacentes de v (que se llaman w) hacia v. Ciclo es una lista inicialmente vacía }

<u>Algoritmo</u> HallarCiclo(G, v, ciclo)
  encontre ← falso
  mientras hay adyacentes para considerar y no encontre hacer
      w ← siguiente adyacente de v en G
      encontre ← HallarCamino(G, w, v, camino)
 finmientras
 si encontre entonces
    ciclo.addLast(v)
    para cada vertice x de camino hacer
      ciclo.addLast(x)
    finpara
 finsi $T_{hallarciclo}(n,m) = O(grado(v)*(n+m) + n)$

DFS con marca y desmarca

Dado un digrafo pesado con números reales, permite hallar un camino de costo mínimo entre dos vértices origen y destino computando el camino y su costo (entendido como la suma de los pesos de los arcos).

El tiempo de ejecución para un grafo que tiene todos los arcos entre cada par de nodos es O(n!) (en la práctica esto es mucho menos, porque n! se da en el peor caso que es cuando todos los vértices están conectados con todos los otros vértices)

Al comienzo, *camino_actual* es una lista vacía y *costo_camino_actual* es 0.

El costo del camino es la suma de los pesos de los arcos del camino.

```
Algoritmo HallarCaminoMinimo(G, origen, destino, camino_actual,
        costo_camino_actual, camino_minimo, costo_camino_minimo)
origen.put( Estado, Visitado )
Camino_actual.addLast( origen )
Si origen = destino entonces
    Si costo_camino_actual  < costo_camino_minimo  entonces
        camino_minimo ← camino_actual
        costo_camino_minimo ← costo_camino_actual
Sino
    para cada adyacente v de origen en G hacer
        si v.get( Estado ) = NoVisitado entonces
          HallarCaminoMinimo( G, v, destino, camino_actual,
                costo_camino_actual + peso(origen,v), camino_minimo,
                costo_camino_minimo )
camino_actual.remove( camino_actual.last() )   { backtracking }
origen.put( Estado, NoVisitado )
```

Mantenimiento del costo mínimo

Java tiene pasaje de parámetros por valor.

Por lo tanto, programar código <u>erróneo</u> como:

```
void buscar_camino( ... float costo_actual, Float costo_minimo, …. ) {
          If( costo_actual < costo_minimo ) {
             ….
            costo_minimo = costo_actual;
            ….
          }
 }
```

<u>No funciona</u> incluso usando Float (clase wrapper de float).

Forma correcta del mantenimiento del costo mínimo

Por lo tanto se recomienda usar una clase:

```
class Solucion {
        float costo;
        … setters y getters para el costo…
}
```

O mejor aún:

```
class Solucion<A> {
        A estado;
        … setters y getters para el estado …
}
```

BFS para hallar caminos

```
// Retorna true si hay camino de s a t en G.
// Previo es un mapeo tal que previo.get(v) retorna el nodo previo de v
// en el camino de s a v.
BFSSearch( G : Graph<V,E>, s : Vertex<V>, t : Vertex<V>;
        previo: Map<Vertex<V>, Vertex<V>> ) : boolean
encolar s en Q; marcar s
while Q no está vacía Do
        desencolar x de Q
        if x = t then return true
        for cada adyacente v de x do
                if v no marcado then
                        encolar v en Q
                        marcar v
                        previo.put(v,x)
Return false
```

Nota: Encuentra el camino más corto en cantidad de arcos entre s y t

$T(n,m) = O(n+m)$

Recuperación del camino

```
Recuperar( s, t, previo ) : Lista
x ← t
Mientras x <> null
        apilar x en pila p
        x ← previo.get(x)
Fin mientras
Crear Lista L
Mientras p no vacía
      L.addLast(p.pop())
Retornar L
```

T(n) = O(n) cuando addLast funciona en O(1) con n la longitud del camino (que se puede acotar con la cantidad de vértices del grafo)

Recuperación del camino (versión 2)

```
Recuperar( s, t, previo ) : Lista
x ← t
Crear Lista L
Mientras x <> null
        L.addFirst(x)
        x ← previo.get(x)
Fin mientras
Retornar L
```

T(n) = O(n) cuando con n la longitud del camino (que se puede acotar con la cantidad de vértices del grafo)

Algoritmo de Dijkstra

- Suponga un digrafo G tal que cada arco tiene costo no negativo, Dijkstra computa los caminos de costo mínimo desde un vértice *a* a todos los otros vértices de G.
- El vértice *a* se conoce como la *fuente*.
- El costo del camino es la suma de los pesos de los arcos del camino.
- El algoritmo mantiene un conjunto S con los vértices cuyo camino con la distancia más corta es conocida.
- S inicialmente está vacío.
- En cada paso se agrega a S el vértice *u* cuya distancia a la fuente es tan cercana como es posible.
- El algoritmo termina cuando S contiene todos los vértices.
- La salida del algoritmo son dos mapeos D : V → Float y P: V → V tal que:
  - D(v) es la distancia a v desde la fuente
  - P(v) es el vértice anterior a v en el camino desde la fuente a v.

<u>Algoritmo</u> Dijkstra

<u>Entrada:</u> G : digrafo simple conexo con todos lo pesos positivos y a : Vertice

{ G tiene vértices $a=v_0, v_1, \ldots, v_n$ y pesos $w(v_i,v_j)$ donde $w(v_i,v_j)=\infty$ si (v_i,v_j) no es un arco en G }

<u>Salida:</u> D : mapeo de vértice en float y P : mapeo de vértice en vértice

```
for i := 1 to  n do begin
        D(i) := ∞
        P(i) := 0
end
D(a) := 0
S := ∅
for i := 1 to n do begin
        u := un vértice no en S con D(u) mínimo
        S := S U { u }
        for cada vértice v adyacente a u y que no está en S do
           if D(u) + w(u,v) < D(v) then begin
                D(v) := D(u) + w(u,v)
                P(v) := u
           end
end
return (P,D)
```

Algoritmo Dijkstra

Entrada: G : digrafo simple conexo con todos lo pesos positivos y a : Vertice

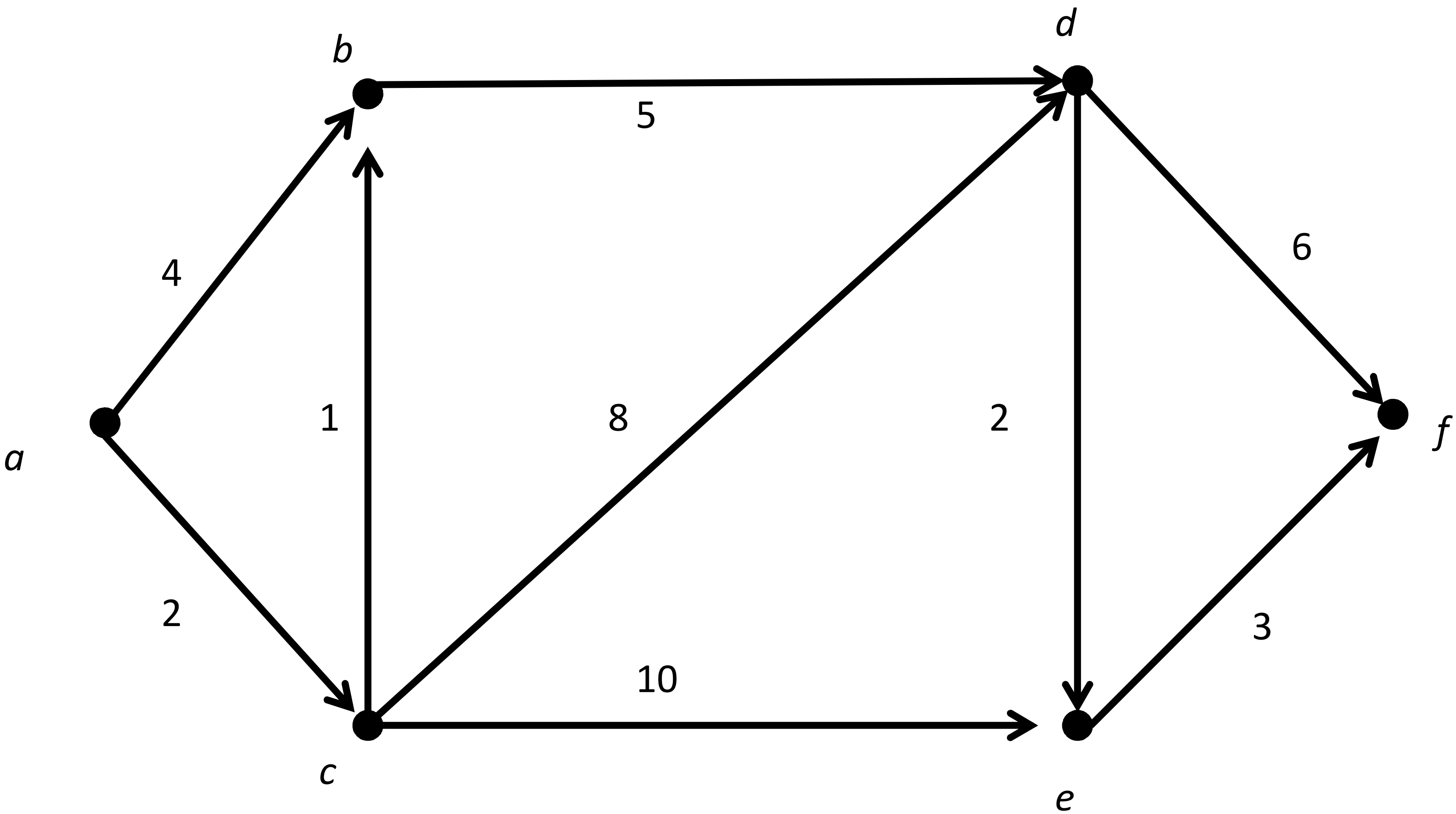
{ G tiene vértices $a=v_0, v_1, \ldots, v_n$ y pesos $w(v_i,v_j)$ donde $w(v_i,v_j)=\infty$ si (v_i,v_j) no es un arco en G }

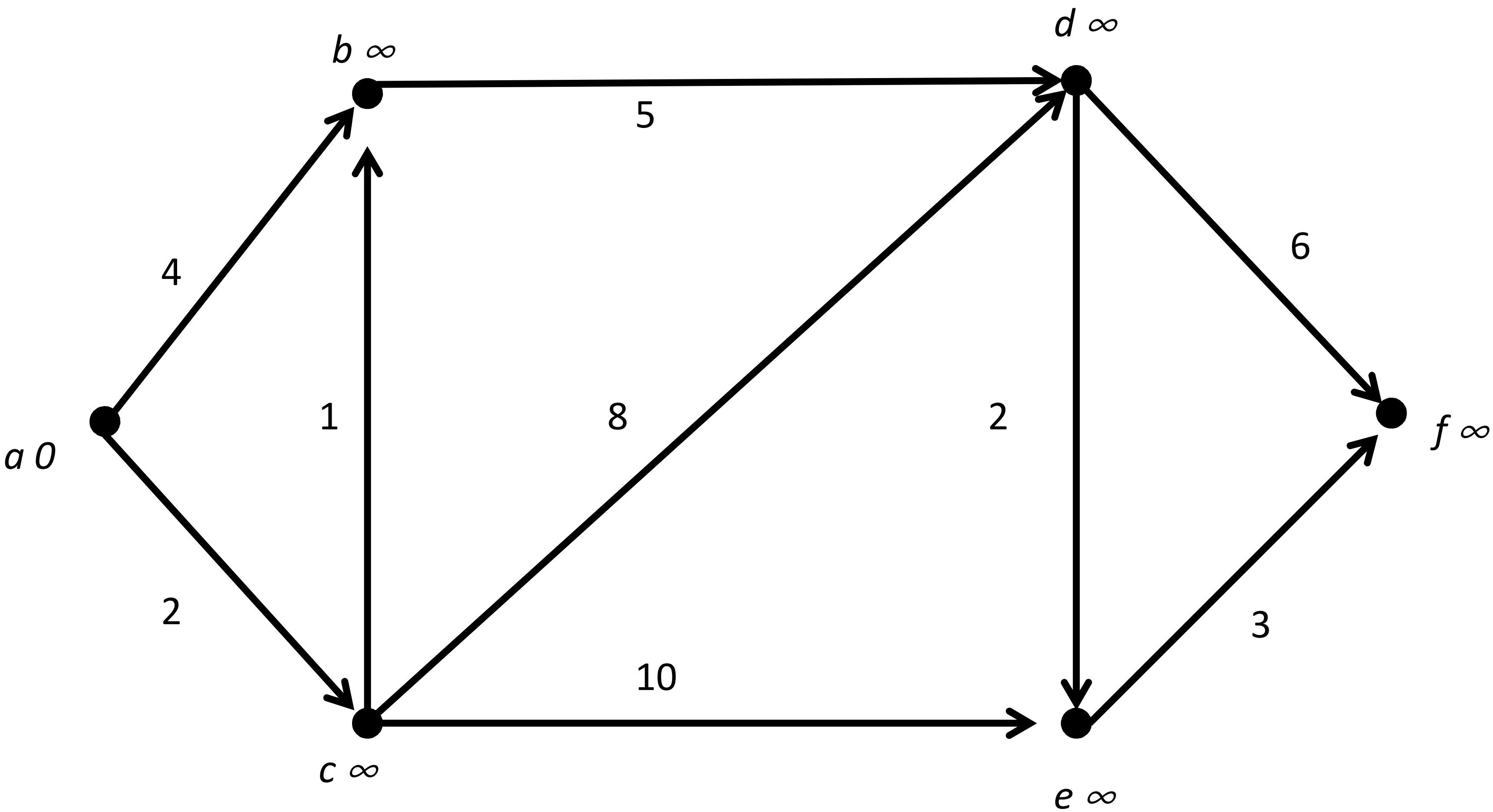
Salida: D : mapeo de vértice en float y P : mapeo de vértice en vértice

$T_{Dijkstra}(n) = O(n^2)$ si G tiene n vértices y está representado con matriz

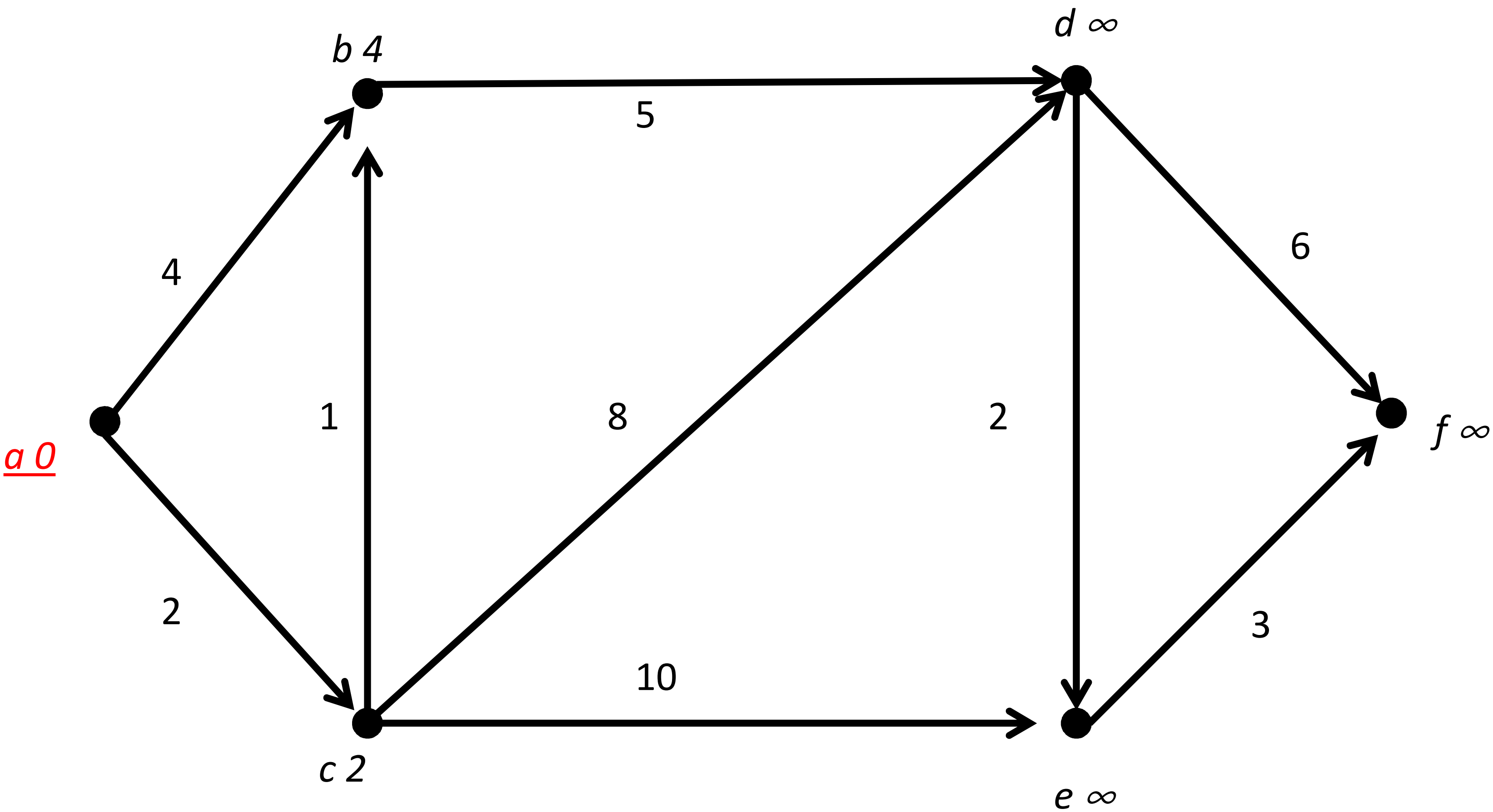
$T_{Dijkstra}(n,m) = O(m \log(n))$ (son m actualizaciones a una cola con prioridades adaptable de n elementos), si G está representado con lista de adyacencias y get y put del mapeo tienen O(1).

```
for i := 1 to  n do begin
        D(i) := ∞
        P(i) := 0
end
D(a) := 0
S := ∅
for i := 1 to n do begin
        u := un vértice no en S con D(u) mínimo
        S := S U { u }
        for cada vértice v adyacente a u y que no está en S do
           if D(u) + w(u,v) < D(v) then begin
                   D(v) := D(u) + w(u,v)
                   P(v) := u
           end
end
return (P,D)
```


b
d
5
4
6
1
8
2
a
f
2
3
10
c
e



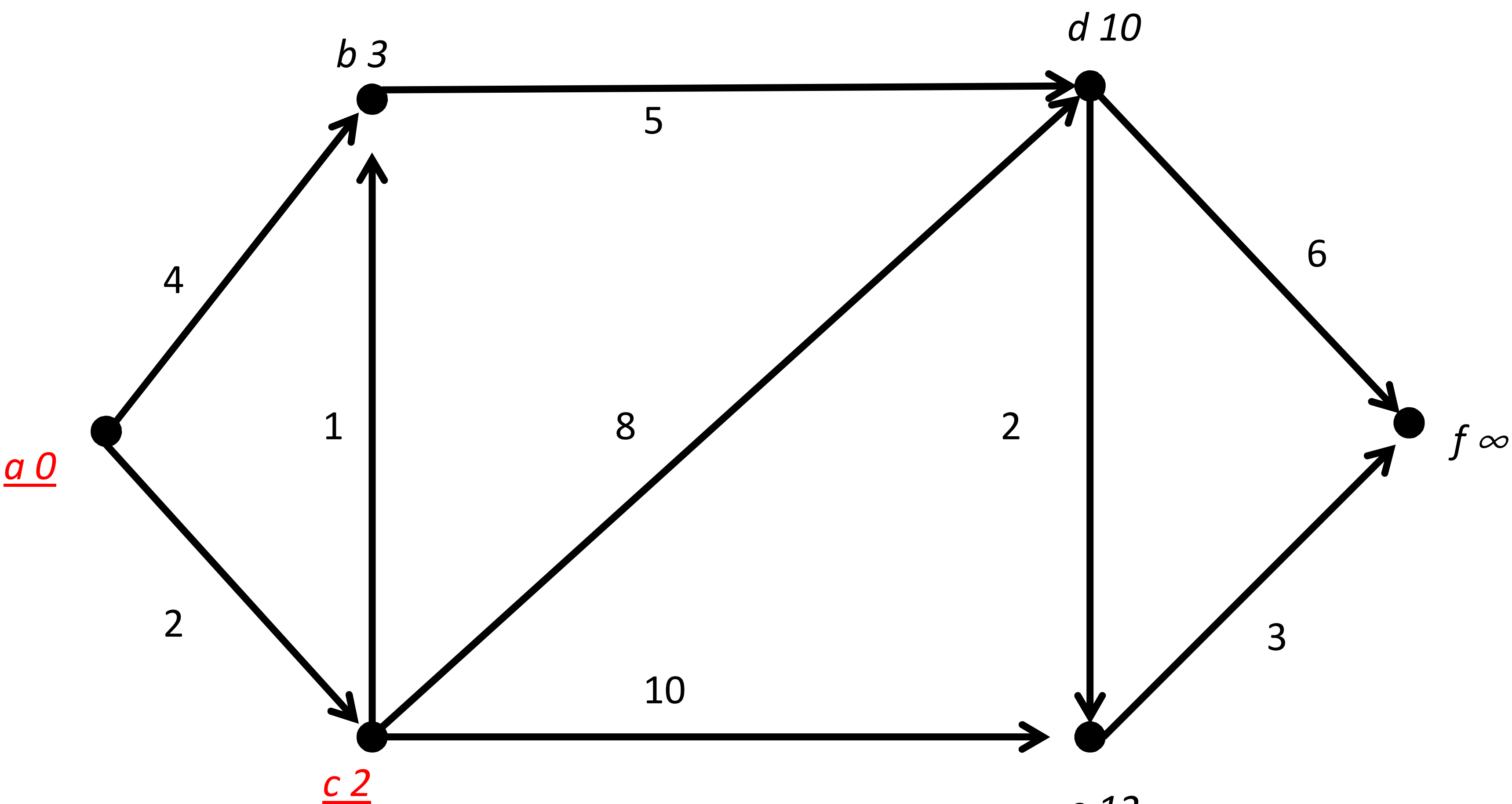
S = ∅
El primer vértice *u* que encuentra el algoritmo es *a* puesto que
D(a) = 0 y todos los otros D(v) = ∞.



u = *a*

Actualiza *a* b y a *c* con D(*b*) = 4 y D(*c*) = 2. S = {*a*}. P(b) = 1 y P(c) = 1.

En la siguiente iteración el mínimo de D será c.

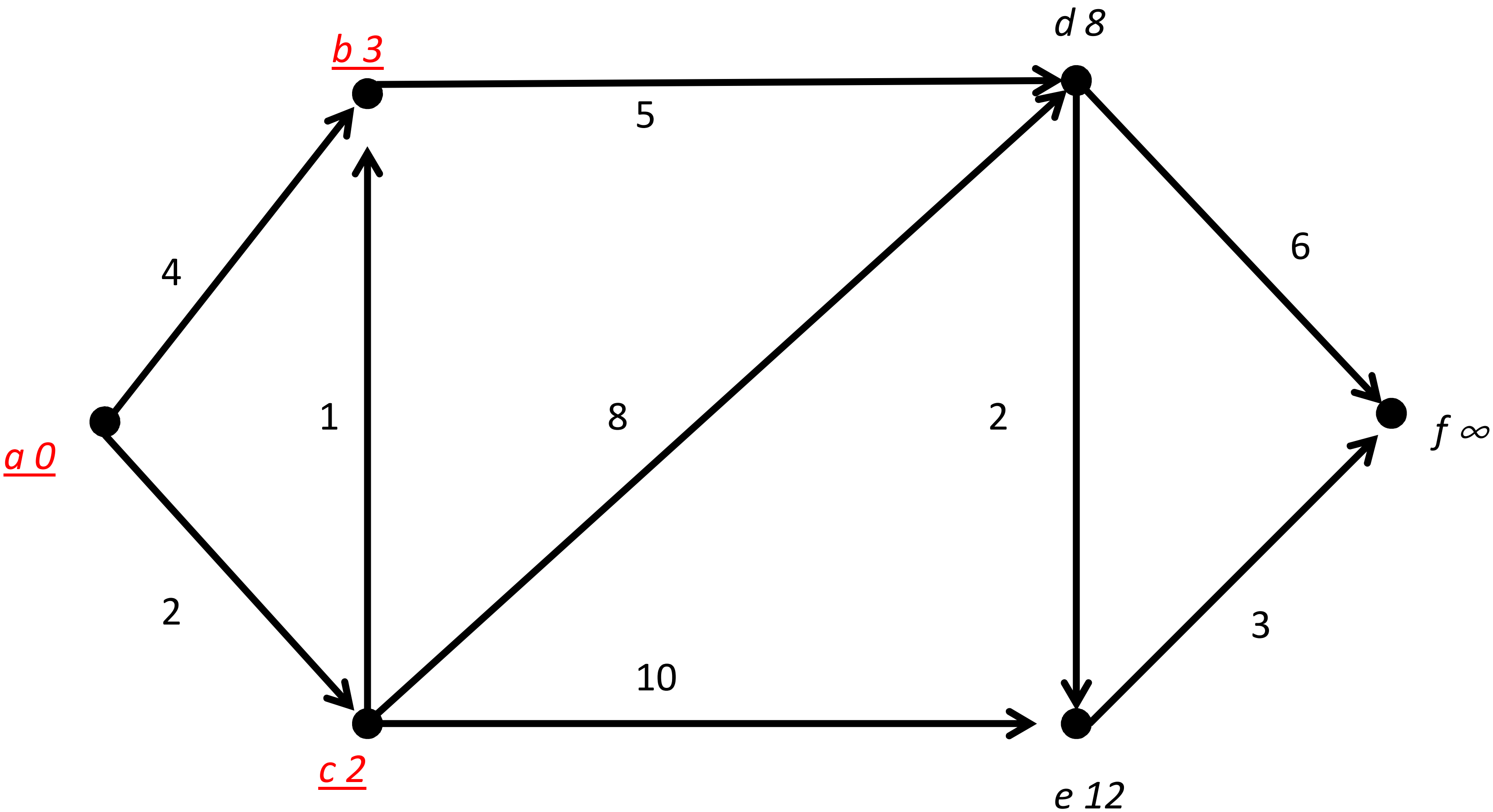


u = c

Actualiza a *b, d y e* con D(*b*) = 3, D(d) = 10 y D(*e*) = 12.

S = {*a,c*}. P(b) = 2, P(d) = c, P(e) = c.

En la siguiente iteración el mínimo de D será b.

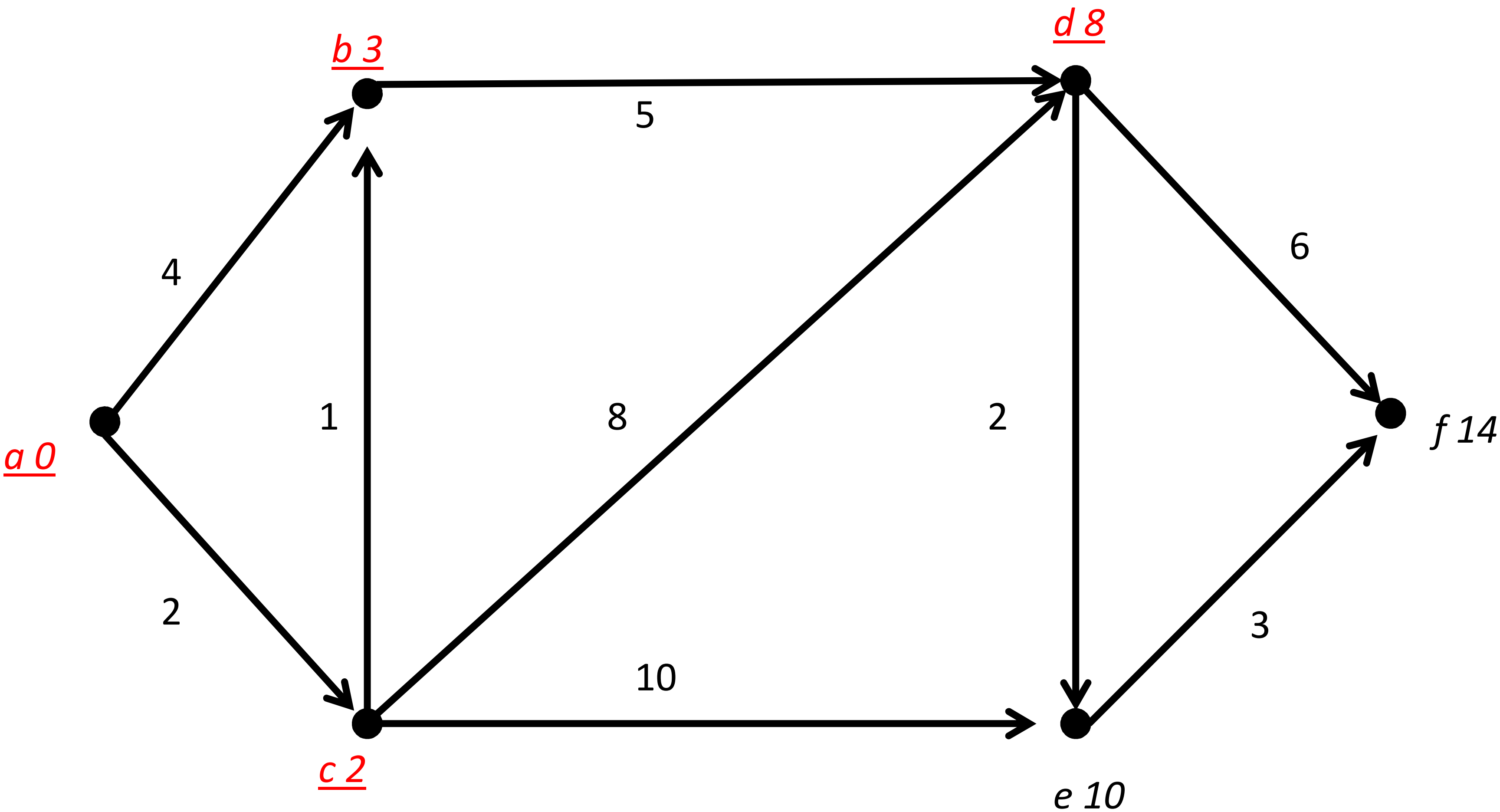


u = b

Actualiza a *d* con D(*d*) = 8,

S = {*a, b, c*}. P(d) = 3

En la siguiente iteración el mínimo de D será d.

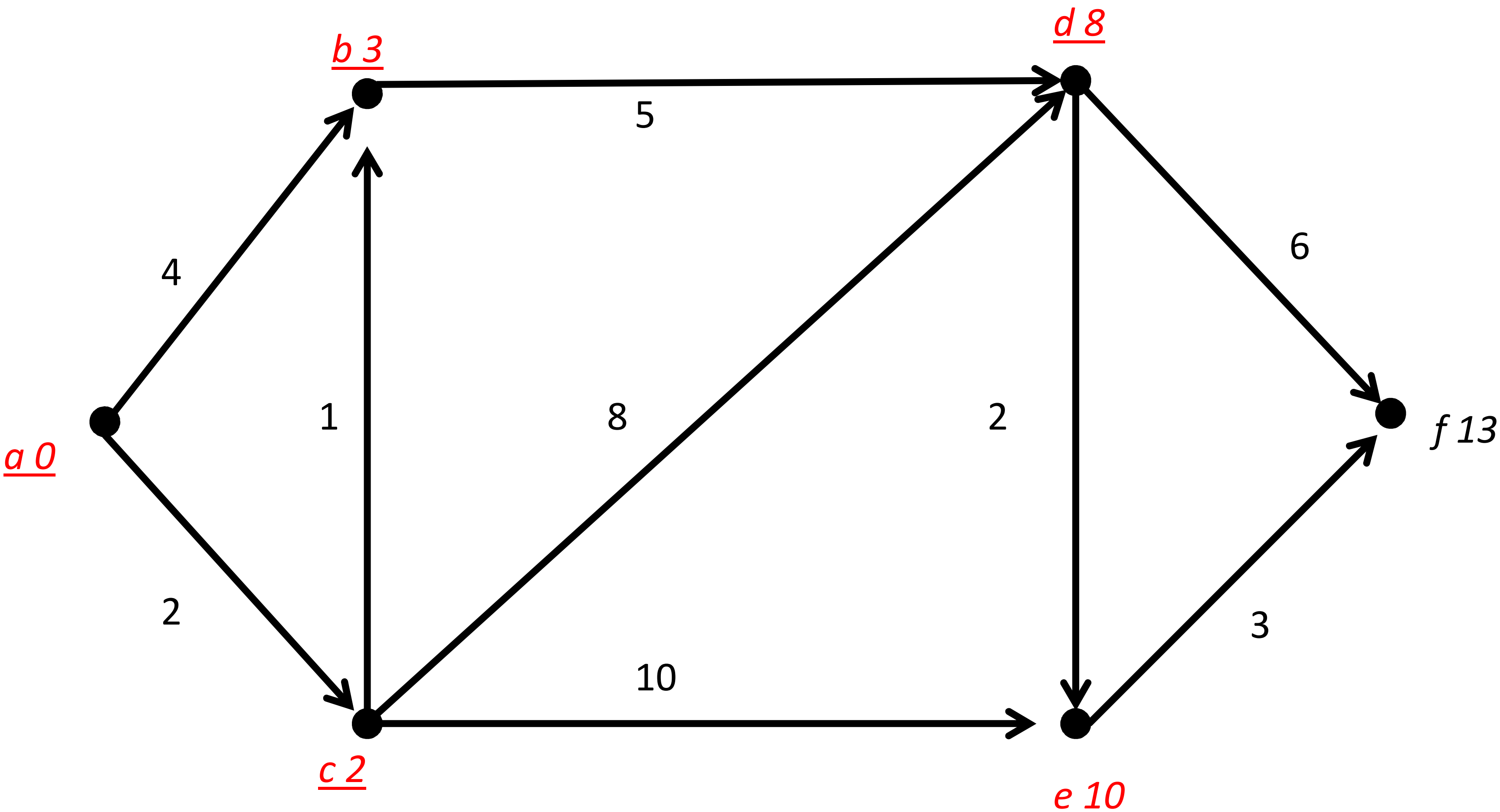


u = d

Actualiza a *e* con D(*e*) = 10 y a f con D(f) = 14. P(e) = d y P(f) = d.

S = {*a*, *b*, *c*, *d*}.

En la siguiente iteración el mínimo de D será e.

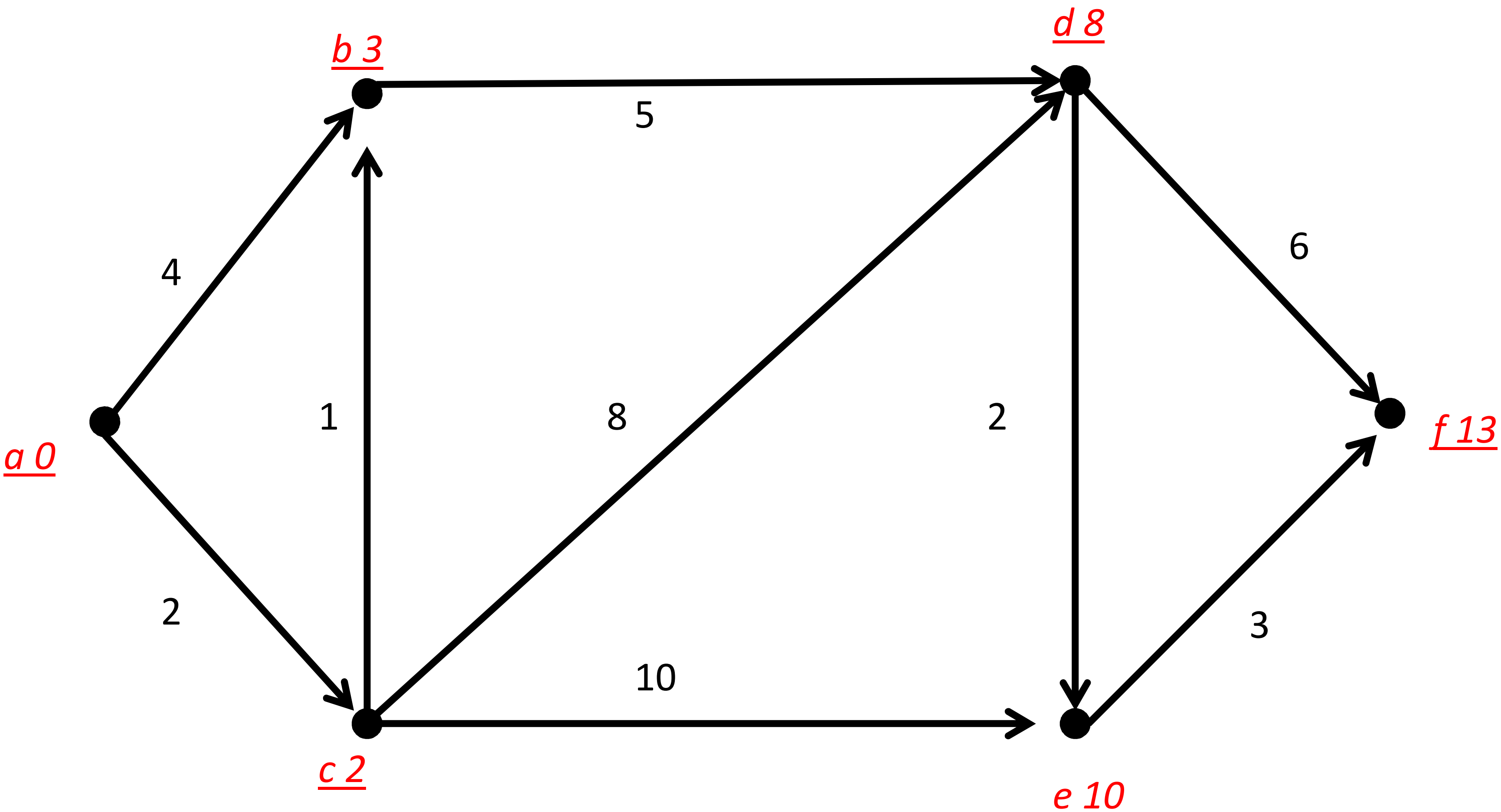


u = e

Actualiza a *f* con D(*f*) = 13. P(*e*) = *d* y P(*f*) = *e*.

S = {*a, b, c, d, e*}.

En la siguiente iteración el mínimo de D será f.



u = f

Como f no puede alcanzar a ningún vértice, no se actualiza ninguno.

S = {*a, b, c, d, e, f*}.

Esta es la iteración final.

Recuperación del camino en Dijkstra

A partir del mapeo *P*, dado un vértice *v* destino, el camino de costo mínimo de la fuente a *v* es *D*(*v*) y el camino en orden inverso es es *v*, *P*(*v*), *P*(*P*(*v*)), *P*(*P*(*P*(*v*))), …

Algoritmo recuperar(P, destino, a)

cola ← new Cola()

cola.enqueue(a)   { encolo la fuente }

recuperar_aux(P, destino, cola)

retornar cola

Algoritmo recuperar_aux(P, v, cola)

anterior ← P(v)

Si anterior = 0 entonces

{ el camino es directo (xq el anterior es la fuente) y terminamos }

Sino

recuperar_aux(P, anterior, cola)

cola.enqueue(anterior)

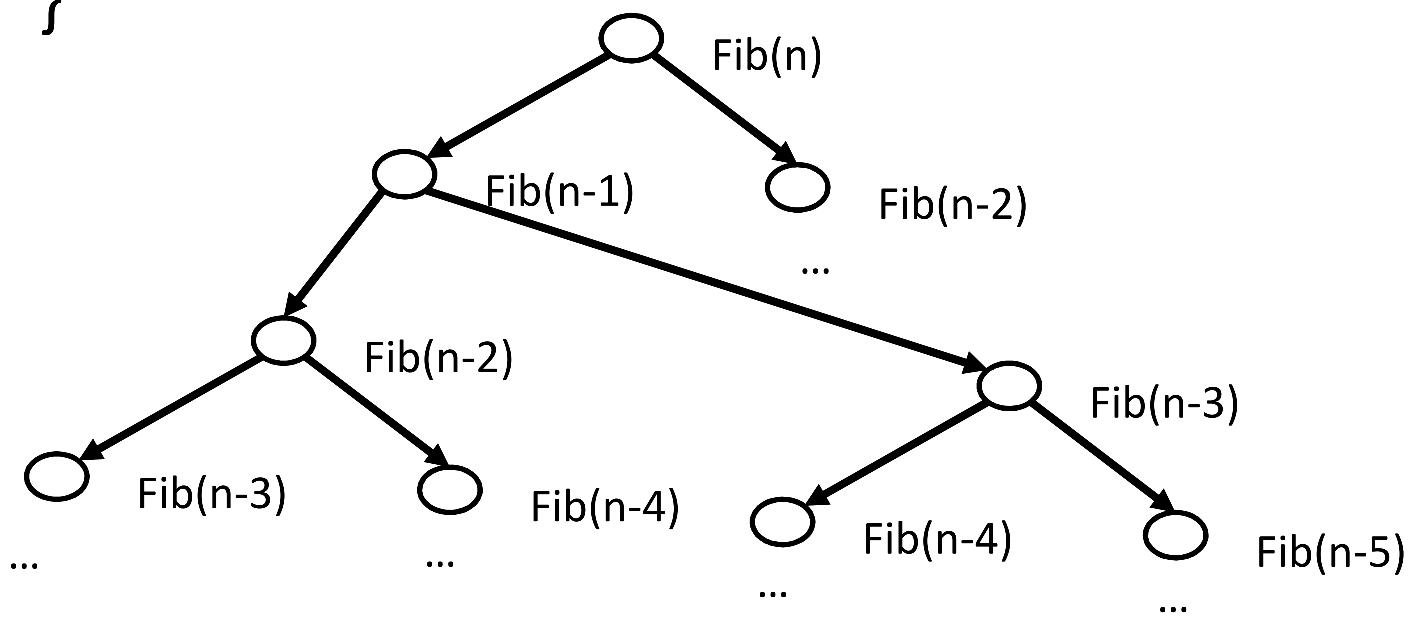
Nota: $T_{recuperar}(n) = O(n)$. Nota: La recuperación se puede plantear en forma iterativa usando un while y una pila.

Programación dinámica

La programación dinámica es un método para resolver un problema dividiéndolo en una colección de subproblemas menores, resolviendo esos subproblemas sólo una vez, y almacenando sus soluciones usando una estructura de datos en memoria principal (e.g. arreglo, mapeo, etc.)

Fibonacci tradicional

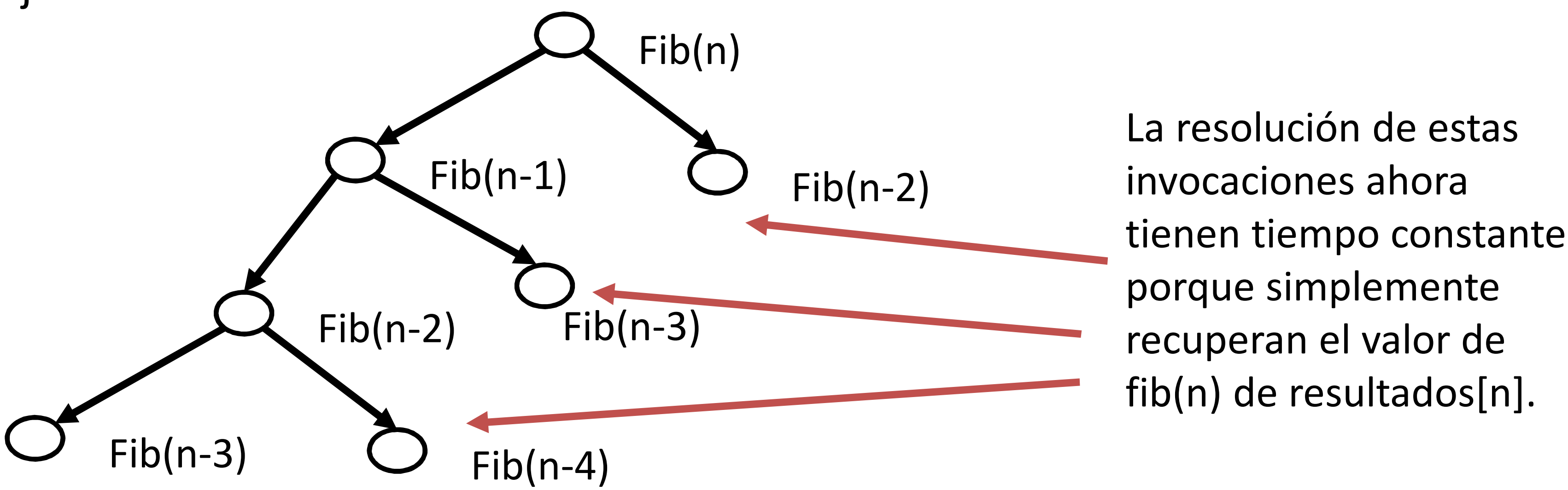
```
int fib(int n) {
  if (n==0 || n==1) return n;
  else return fib(n-1) + fib(n-2);
}
```



Considerando el árbol de llamadas recursivas, se puede ver que rl orden del tiempo de ejecución de esta función es exponencial porque vuelve a recalcular muchos resultados intermedios.

Fibonacci con programación dinámica

```
int fib(int n, int [] resultados) {
        if (resultados[n] != 0) return resultados[n];
        if (n==1){ resultados[1]=1; return resultados[1]; }
        else { resultados[n] = fib(n-1)+fib(n-2); return resultados[n]; }
}
```



…

Al usar un arreglo *resultados* para almacenar los resultados intermedios ya calculados, se puede bajar el orden a n.

Hacia el algoritmo de Warshall: Cómputo de la clausura transitiva R* de una relación binaria R

- Una relación binaria R es transitiva cuando aRb y bRc implica aRc.
- Dado un grafo que representa una relación R, el algoritmo de Warshall permite computar la clausura transitiva de R, notada como R*.
- La clausura transitiva de R es la relación transitiva más pequeña que contiene a R.
- Cuando R es representada con un grafo G dirigido, si R no es transitiva, entonces G no contiene todos los arcos para los vértices que pueden ser unidos mediante caminos.
- Ej: R = { (1,2), (2,3) } entonces R* = R U {(1,3)} pues es posible ir de 1 a 3 (pasando por 2).

Cómo calcular R*

- Si la relación R entre n elementos se representa con una matriz booleana de nxn (booleana quiere decir formada por 1s y 0s), entonces R^n (es decir, R x R x R x…xR realizado n veces, con "x" representado el producto booleano de matrices).
- La clausura R* se calcula como R U R^2 U R^3 U …. U R^n, donde U representa el join-booleano (or-booleano componente a componente) entre matrices.

EXAMPLE 7 Find the zero-one matrix of the transitive closure of the relation R where

$$\mathbf{M}_R = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

Solution: From Theorem 3, it follows that the zero-one matrix of R^* is

$$\mathbf{M}_{R^*} = \mathbf{M}_R \vee \mathbf{M}_R^{[2]} \vee \mathbf{M}_R^{[3]}.$$

Since

$$\mathbf{M}_R^{[2]} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{M}_R^{[3]} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

it follows that

$$\mathbf{M}_{R^*} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \vee \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \vee \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

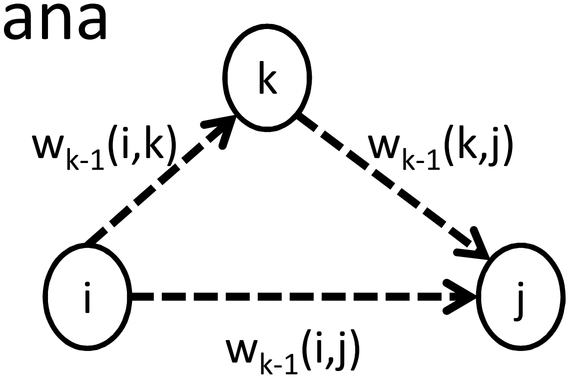
□

```java
// Realiza la clausura transitiva de una matriz booleana
// cuadrada de tamaño n en O(n^4)
public static void clausura_transitiva(
        int[][]a, int n, int [][] a_star){
  int [][] prod = new int[n][n];
  int [][] prod_2 = new int[n][n];

  copiar(a, prod, n, n);
  for(int i=1; i<n; i++) { // n-1 iteraciones
       producto(prod, a, n, n, n, prod_2); // O(n^3)
       hacer_join(prod_2, prod, n, n);     // O(n^2)
  }
  copiar(prod, a_star, n, n);  // O(n^2)
}

// Copia la matriz a en la b
public static void copiar(int [][]a,
        int [][]b, int n, int m) {
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            b[i][j] = a[i][j];
}
```

```java
// Hace el join booleano de las matrices a y b de n x m.
// Hacer max(x,y) es lo mismo que hacer x or y
public static void hacer_join(int [][]a, int [][]b, int n, int m) {
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            b[i][j] = Math.max(b[i][j], a[i][j]);
}
```

```java
// Realiza el producto booleano de las matrices a y b de n x p y p x m resp.
// hacer max(x,y) es lo mismo que hacer x or y
// hacer min(x,y) es lo mismo que hacer x and y
public static void producto(int [][] a, int [][] b,
    int n, int p, int m, int [][] c) {
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++) {
            c[i][j] = 0;
            for(int k=0; k<p; k++)
                c[i][j] = Math.max(c[i][j], Math.min(a[i][k], b[k][j]));
        }
}
```

<u>Estrategia:</u> Para cada vértice k, para cada par de vértices (i,j) ver si puedo conectar i con j a través de k y si es así, agregar (i,j) a la clausura transtiva.



Procedure Warshall(M_R : Matriz booleana de n x n)

W := M_R

for k := 1 to n do

for i := 1 to n do

for j := 1 to n do

w(i,j) := w(i,j) or (w(i,k) and w(k,j))

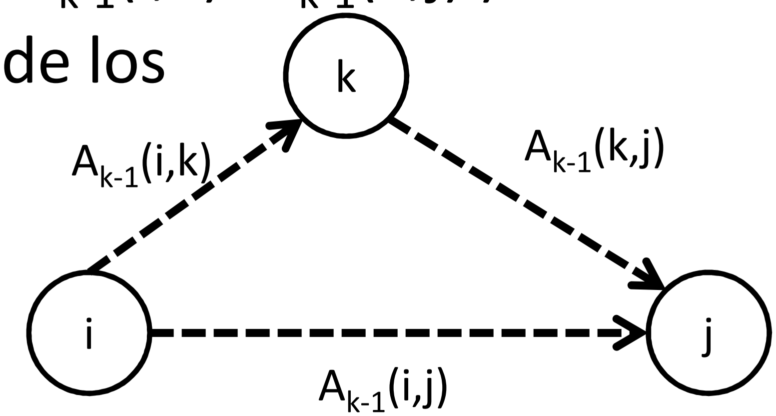
End { W = [w_{ij}] es M_{R*} }

$$T_{warshall}(n) = O(n^3)$$

```java
public static void warshall( int [][]a, int n, int [][]w) {
   copiar(a, w, n, n);    // O(n^2)
   for(int k=0; k<n; k++ )    // n iteraciones
     for(int i=0; i<n; i++)  // n iteraciones
        for(int j=0; j<n; j++)    // n iteraciones
            w[i][j] = Math.max(w[i][j],
                 Math.min(w[i][k], w[k][j])); // O(1)
    // El procedimiento tiene O(n^3)
}
```

Algoritmo de Floyd: Caminos mínimos

- Dado un digrafo pesado G=(V,A) donde cada arco tiene un peso numérico no negativo.
- Queremos calcular los caminos de costo mínimo de todos los vértices a todos los vértices.
- Supongamos que C(i,j) es peso del arco (i,j) de A.
- Usaremos una matriz A de nxn tal que inicialmente A(i,j)=C(i,j), o ∞ si no hay arco entre i y j
- En la iteración k-ésima veremos si actualizamos a A de acuerdo a $A_k(i,j) = \min(A_{k-1}(i,j), A_{k-1}(i,k)+A_{k-1}(k,j))$.
- P(i,j) almacenerá a k (i.e., uno de los vértices intermedios en el camino de i a j.

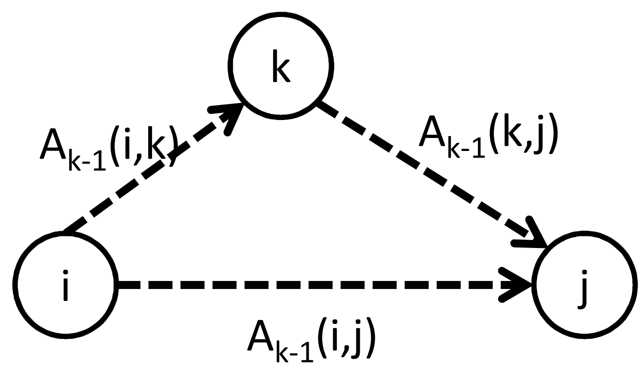


Algoritmo Floyd

Para i $\leftarrow$ 1..n hacer
    para j $\leftarrow$ 1..n hacer
        si hay arco (i,j) entonces A(i,j) $\leftarrow$ C(i,j)
        sino A(i,j) $\leftarrow \infty$
        P(i,j) $\leftarrow$ 0 { por defecto el camino es directo }

Para i $\leftarrow$ 1 .. n hacer
    A(i,i) $\leftarrow$ 0

Para k $\leftarrow$ 1..n hacer
    para i $\leftarrow$ 1..n hacer
        para j $\leftarrow$ 1..n hacer
            si a(i,k) + a(k,j) < a(i,j) entonces
                a(i,j) $\leftarrow$ a(i,k) + a(k,j)
                p(i,j) $\leftarrow$ k

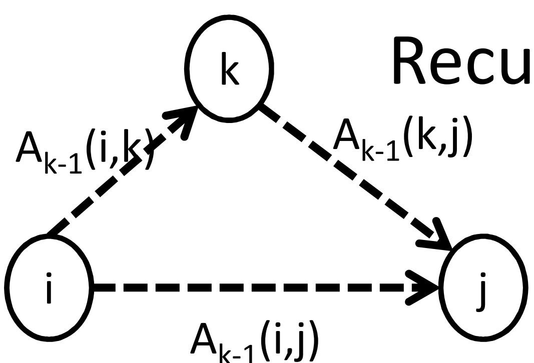


$T_{Floyd}(n) = O(n^3)$

Recuperación del camino en Floyd

{ Recupera el camino de i a j y lo almacena en cola que inicialmente debe estar vacía. }

```
Algoritmo RecuperarCamino( P, i, j, cola )
        k ← P(i, j)
        si k ≠ 0 entonces
                RecuperarCamino( P, i, k, cola )
                cola.enqueue( k )
                RecuperarCamino( P, k, j, cola )
```



$T_{RecuperarCamino}(n) = O(n)$ para un grafo de n vértices.

Bibliografía

- Capítulo 13 de M. Goodrich & R. Tamassia, Data Structures and Algorithms in Java. Fourth Edition, John Wiley & Sons, 2006.