M. Caramihai, © 2020

STRUCTURI DE DATE & ALGORITMI

CURS 11

Programarea dinamica.

Algoritmul *backtracking*

Programarea dinamica (PD)

- Programarea dinamica reprezinta o paradigma de proiectarea principiala a algoritmilor:
  - Rezolvarea iterativa a unor sub-probleme (mai mici) ceea ce permite (prin combinare) rezolvarea problemei globale.
- Principii:
  - Problema este “sparta” in subprobleme
    - Cel mai dificil
    - Sub-problemele sunt *shared*
    - Solutia optima de rezolvare a sub-problemei contribuie la rezolvarea problemei globale (optimalitatea sub-problemei)
  - Solutiile se calculeaza la nivelul sub-problemelor
  - Solutiile calculate se combina in solutii pentru sub-problele “mai mari”

Elemente de PD

- **Sub-structura optimala:**
  - O problema prezinta substructuri optimale (optimal substructure) daca o solutie optimala (globala) contine solutii optimale pentru *sub-probleme*.
- **Suprapunerea sub-problemelor *(Overlapping subproblems)*:**
  - Multe sub-probleme *share* sub-sub-probleme.

Substructura optimala

- Q problema prezinta substructuri optimale (optimal substructure) daca o solutie optimala (globala) contine if solutii optimale pentru *sub-probleme*.
- Astfel, o solutie optimala globala se poate construi pe baza solutiilor optimale aferente *sub-problemelor.*
- Solutiile sub-problemelor sunt parti ale solutiei finale.
- Exemplu: *Longest Common Subsequence* -

  LCS contine solutiile optimale la prefixele pentru cele doua secvente de intrare.

Problema...

- Exemplu: studiul similaritatii dintre lanturile AND (bioinformatica).
- Lanturile AND sunt considerate ca siruri de litere; se compara apoi similaritatile dintre siruri.
- Exemplu: X = ABCBDAB, Y=BDCABA
- Subsecventele pot fi: ABA, BCA, BCB, BBA

  BCBA

  BDAB etc.

…dar *Longest Common Subsequences (LCS)* este **BCBA** si **BDAB**.

- Cum poate fi identificat LCS in mod eficient?

Brute Force

- Daca |X| = m, |Y| = n, atunci exista 2^m subsecvente ale lui X; fiecare dintre ele trebui comparata cu Y (n comparari)

- Astfel timpul de rulare pentru algoritmul *brute-force* este $O(n\ 2^m)$

  → nu este practic pentru secvente lungi.

- Problema LCS contine mai multe *substructuri optimale* :

  solutii ale subproblemelor fac parte din solutia finala.

- Subprobleme: “sa se gaseasca LCS pentru perechile de *prefixe* ale lui X si Y”

LCS: substructura optimala

fie $X =< x_1, x_2, \ldots, x_m >$ si $Y =< y_1, y_2, \ldots, y_n >$ doua secvente si fie $Z =< z_1, z_2, \ldots, z_k >$ orice LCS a lui X si Y.

If $x_m = y_n$, then $z_k = x_m = y_n$ si Z_{k-1} este LCS pentru X_{m-1} si Y_{n-1}

If $x_m \neq y_n$, then $z_k \neq x_m$

$\Rightarrow Z$ este un LCS al lui X_{m-1} si Y

If $x_m \neq y_n$, then $z_k \neq y_n$

$\Rightarrow Z$ este un LCS al lui X si Y_{n-1}

LCS: *Setup* pentru *Dynamic Programming*

- Fie X_i, Y_j prefixele lui X si Y de lungime *i* si respectiv *j*.
- Fie *c[i,j]* lungimea lui LCS pentru X_i si Y_j
- Prin urmare, lungimea LCS pentru X si Y va fi *c[m,n]*

Recurenta LCS

- Recurenta este exponentiala (problema !).
- Nu se cunoaste *a priori* durata.
- Pentru a gasi LCS, este necesar a se gasi in prealabil LCS pentru c[i, j-1] si pentru c[i-1, j]

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Algoritmul LCS

- Pentru inceput trebuie gasita lungimea LCS. Ulterior, algoritmul trebuie modificat pentru a gasi chiar LCS.
- Fie X_i, Y_j prefixele lui X si Y de lungime *i* si *j* (dupa cum s'a mentionat anterior)
- Se defineste ***c[i,j]*** = lungimea lui LCS pentru X_i si Y_j

→ LCS pentru X si Y va fi ***c[m,n]***

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{in rest} \end{cases}$$

Solutia recursiva LCS

- Se porneste cu *i = j = 0* (subsiruri goale ale lui x si y)
- Deoarece X_0 si Y_0 sunt subsiruri goale → LCS este totdeauna gol (i.e. *c[0,0] = 0*)
- LCS pentru un sir gol → pentru orice i si j: *c[0, j] = c[i,0] = 0*
- Cand se calculeaza *c[i,j],* se iau in considerare doua cazuri:
  - **Cazul 1:** *x[i]=y[j]*: sirurile X si Y au cate un element comun in plus; lungimea lui LCS X_i si Y_j *este egala cu lungimea LCS a sirurilor* X_{i-1} si Y_{i-1} , plus 1
  - **Cazul 2:** *x[i] != y[j]: daca cele doua siruri nu au nici un element comun, solutia nu se imbunatateste → lungimea* LCS(X_i , Y_j) este identica cu cea de la pasul anterior (i.e. maximum ((LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)))

Exemplu LCS (1)

Fie urmatorul exemplu:

X = ABCB

Y = BDCAB

Care este *Longest Common Subsequence (LCS)* pentru X si Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

Exemplu LCS (2)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Se va loca matricea c[6,5]

Exemplu LCS (3)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0

Exemplu LCS (4)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for j = 0 to n c[0,j] = 0

Exemplu LCS (5)

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0** ↑				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

case i=1 and j=1

A != B

dar c[0,1]>=c[1,0]

→ c[1,1] = c[0,1], si b[1,1] = ↑

Exemplu LCS (6)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0** ↑	**0** ↑			
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

case i=1 and j=2

A != D

dar c[0,2]>=c[1,1]

→ c[1,2] = c[0,2], si b[1,2] = ↑

Exemplu LCS (7)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0** ↑	**0** ↑	**0** ↑		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

case i=1 and j=3

A != C

dar c[0,3]>=c[1,2]

→ c[1,3] = c[0,3] si b[1,3] = ↑

Exemplu LCS (8)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0** ↑	**0** ↑	**0** ↑	**1** ↖	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

case i=1 and j=4

A = A

→ c[1,4] = c[0,3]+1 sib[1,4] = ↖

Exemplu LCS (9)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0** ↑	**0** ↑	**0** ↑	**1** ↖	**1** ←
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

case i=1 and j=5

A != B

acum c[0,5]<c[1,4]

→ c[1,5] = c[1, 4] si b[1,5] = ←

Exemplu LCS (10)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0** ↑	**0** ↑	**0** ↑	**1** ↖	**1** ←
2	**B**	**0**	**1** ↖				
3	**C**	**0**					
4	**B**	**0**					

case i=2 and j=1

B = B

→ c[2, 1] = c[1, 0]+1 si b[2, 1] = ↖

Exemplu LCS (11)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B		0	1 ↖	1 ←			
3	C		0					
4	B		0					

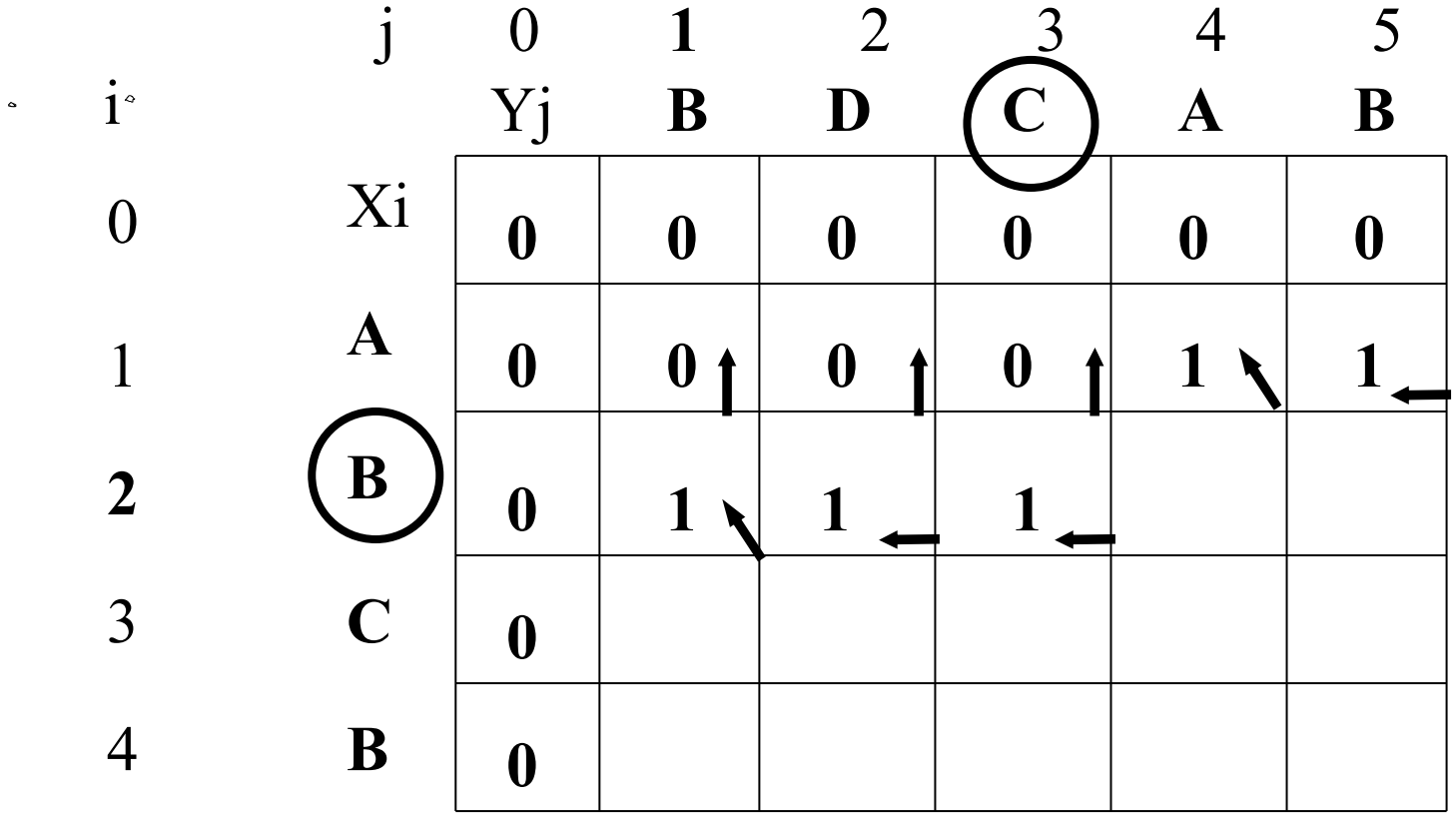
case i=2 and j=2

B != D

si c[1, 2] < c[2, 1]

→ c[2, 2] = c[2, 1] si b[2, 2] = ←

Exemplu LCS (12)



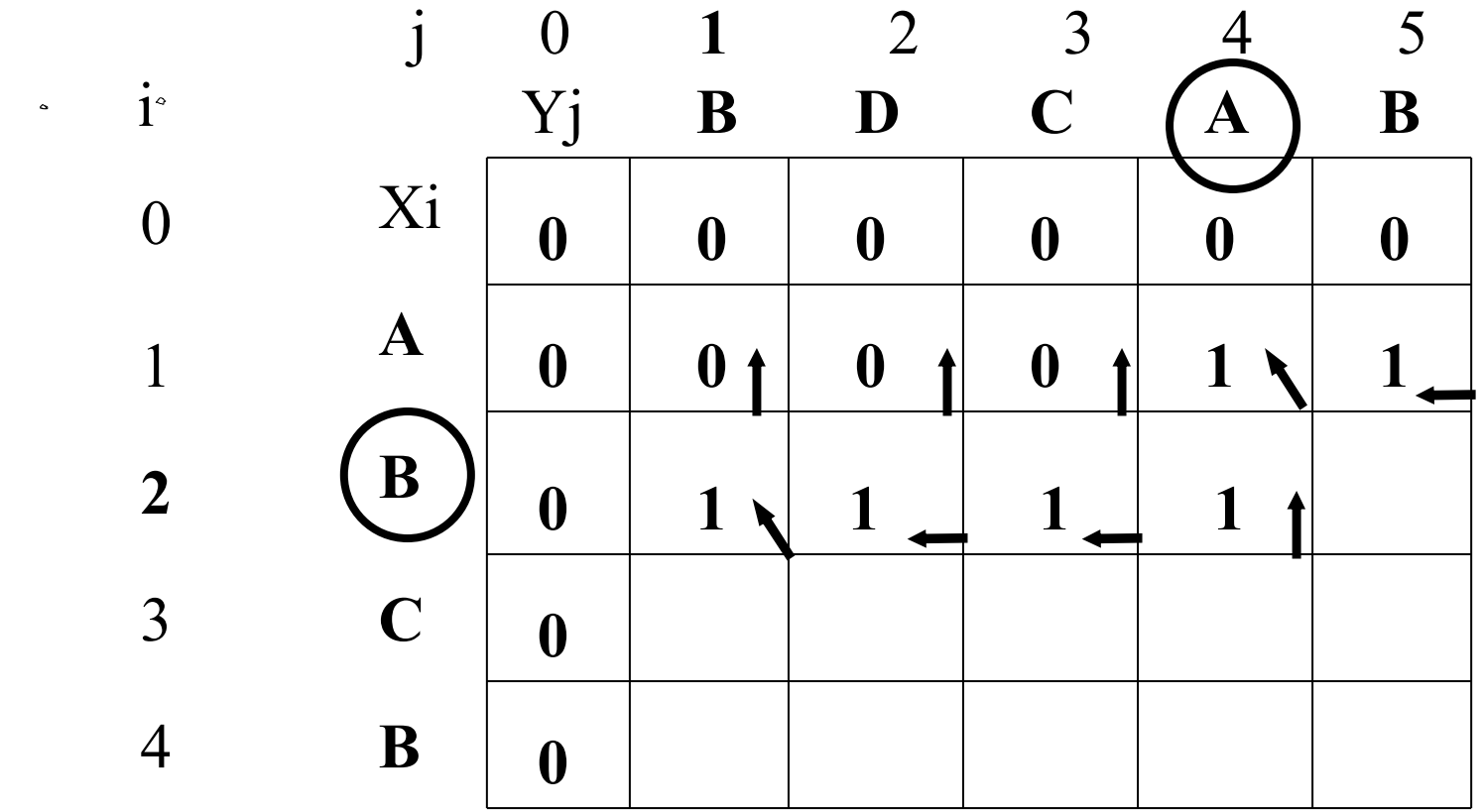
case i=2 and j=3

B != D

si c[1, 3] < c[2, 2]

→ c[2, 3] = c[2, 2] si b[2, 3] = ←

Exemplu LCS (13)



case i=2 and j=4

B != A

si c[1, 4] = c[2, 3]

→ c[2, 4] = c[1, 4] si b[2, 2] = ↑

Exemplu LCS (14)

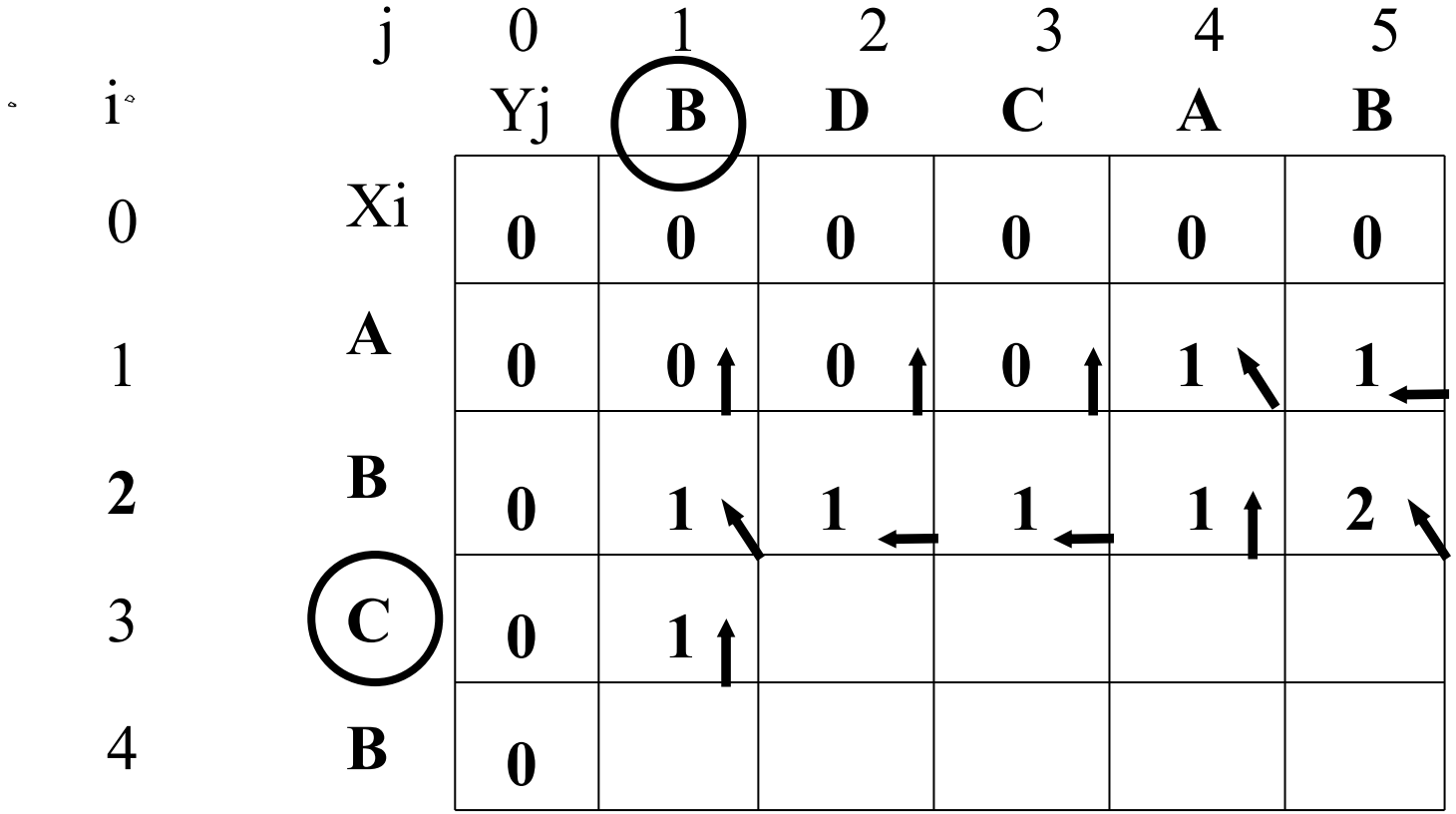
i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	B		0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	C		0					
4	B		0					

case i=2 and j=5

B = B

→ c[2, 5] = c[1, 4]+1 si b[2, 5] = ↖

Exemplu LCS (15)



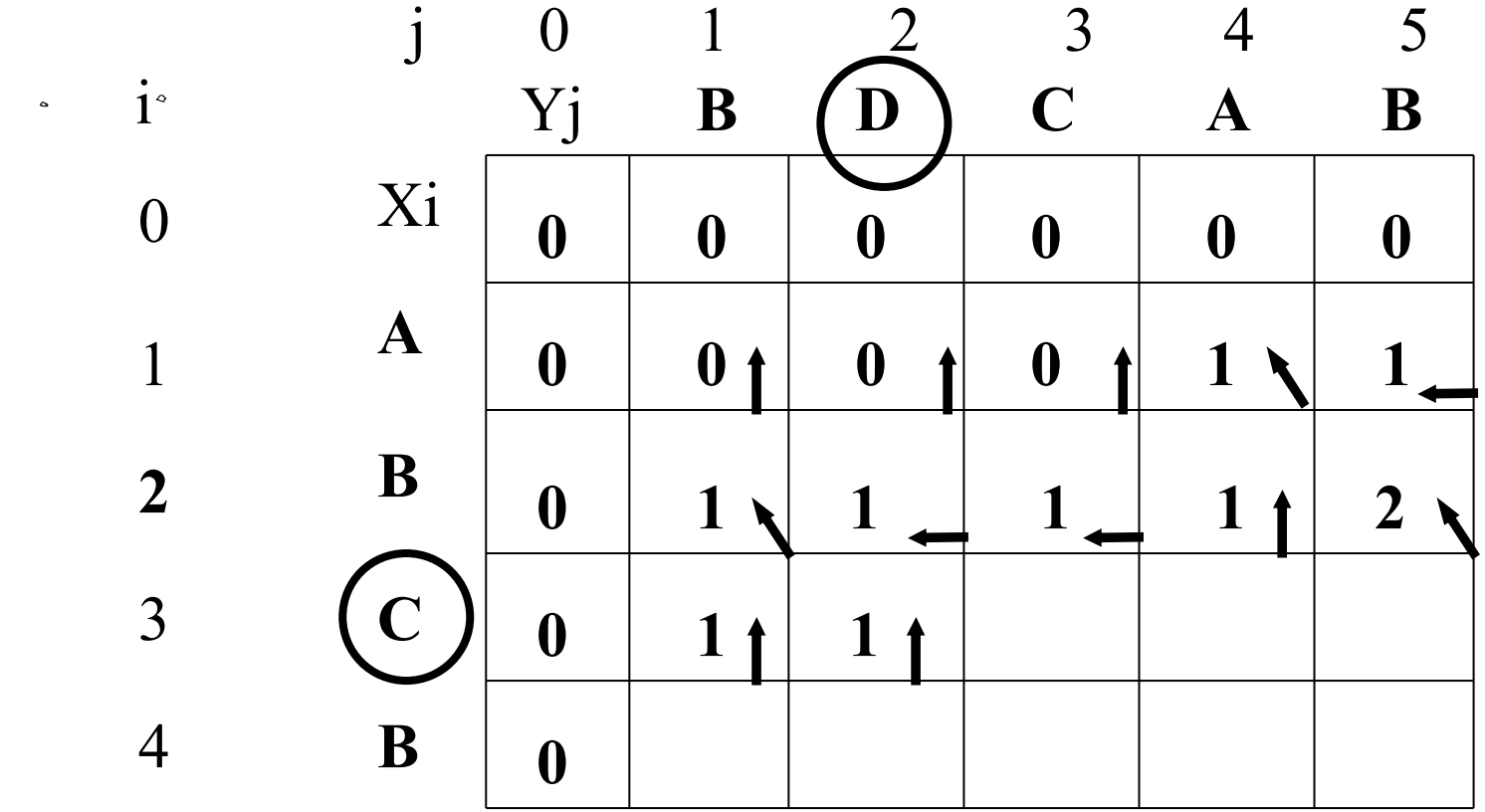
case i=3 and j=1

C != B

si c[2, 1] > c[3,0]

→ c[3, 1] = c[2, 1] si b[3, 1] = ↑

Exemplu LCS (16)



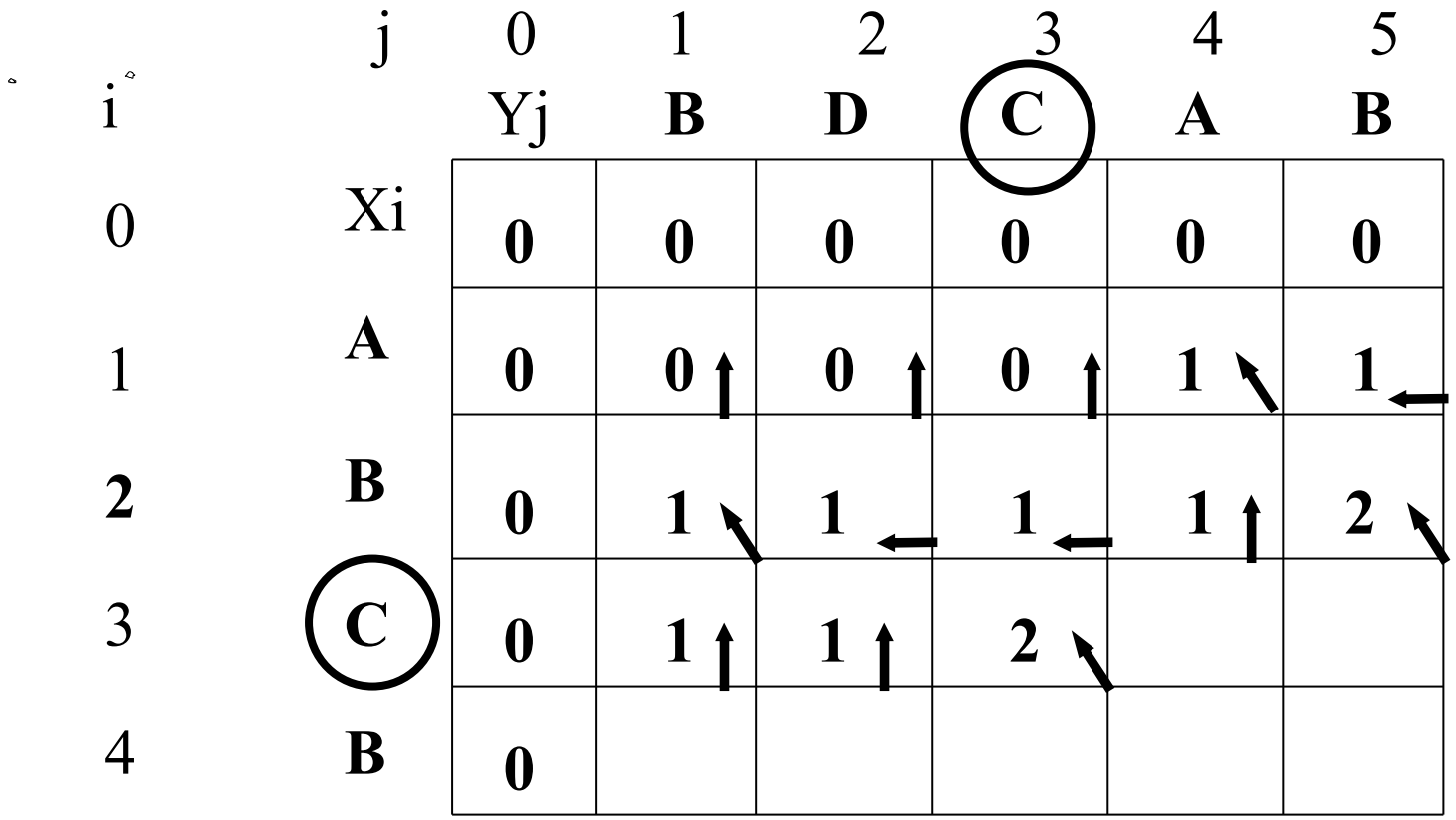
case i=3 and j= 2

C != D

si c[2, 2] = c[3, 1]

→ c[3, 2] = c[2, 2] si b[3, 2] = ↑

Exemplu LCS (17)

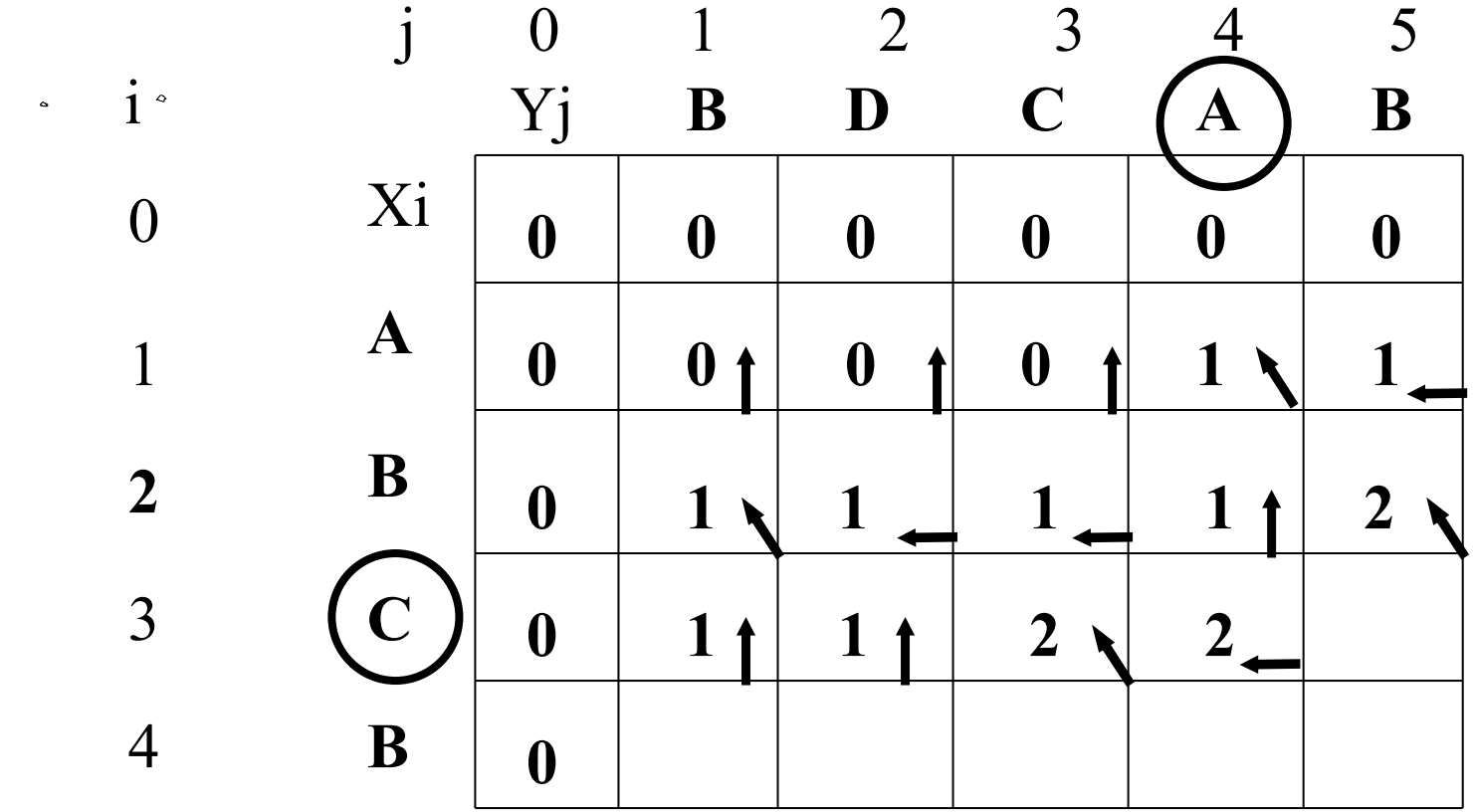


case i=3 and j= 3

C = C

→ c[3, 3] = c[2, 2]+1 si b[3, 3] =

Exemplu LCS (18)



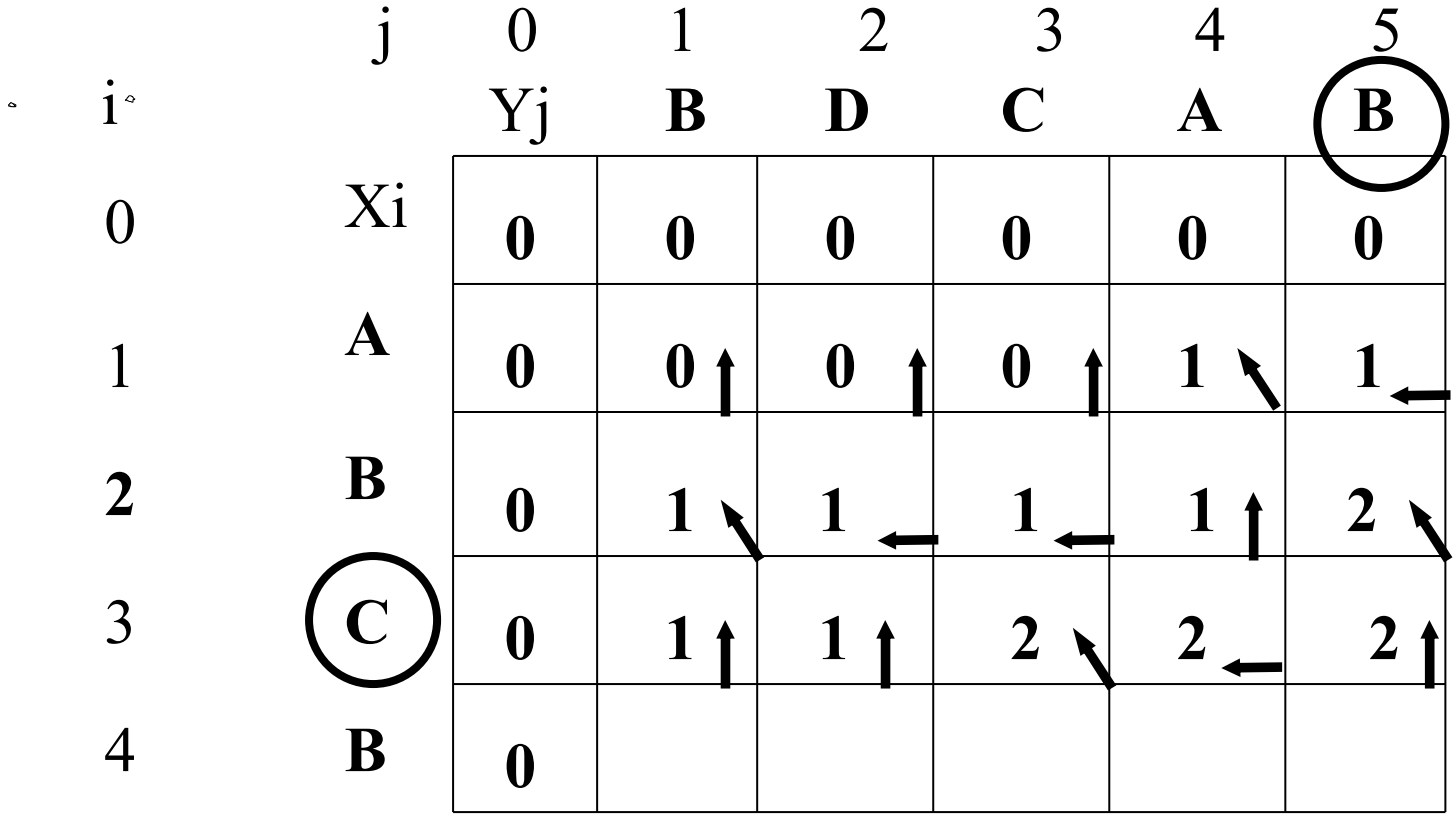
case i=3 and j= 4

C != A

c[2, 4] < c[3, 3]

→ c[3, 4] = c[3, 3] si b[3, 3] = ←

Exemplu LCS (19)



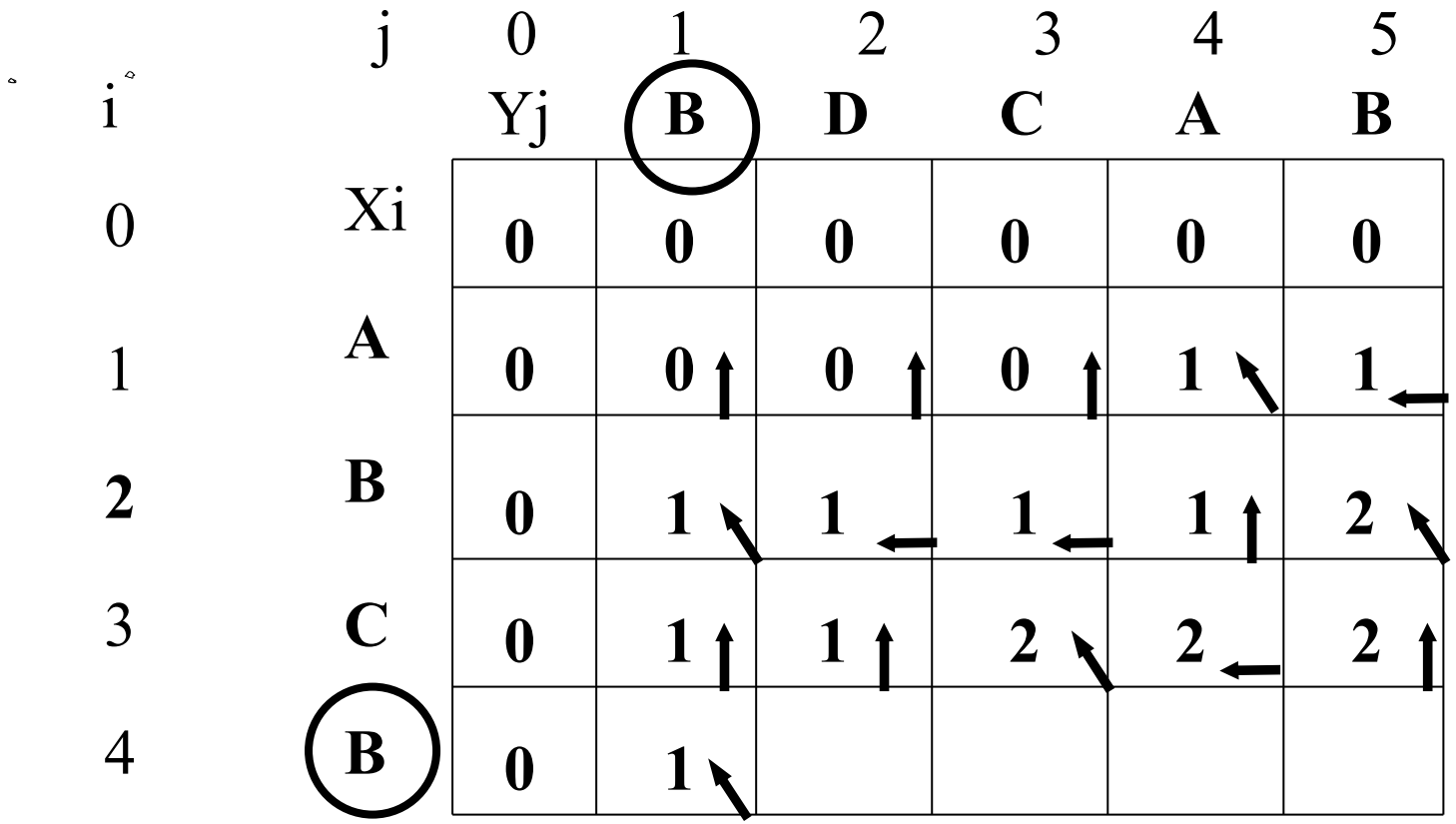
case i=3 and j= 5

C != B

c[2, 5] = c[3, 4]

→ c[3, 5] = c[2, 5] si b[3, 5] = ↑

Exemplu LCS (20)

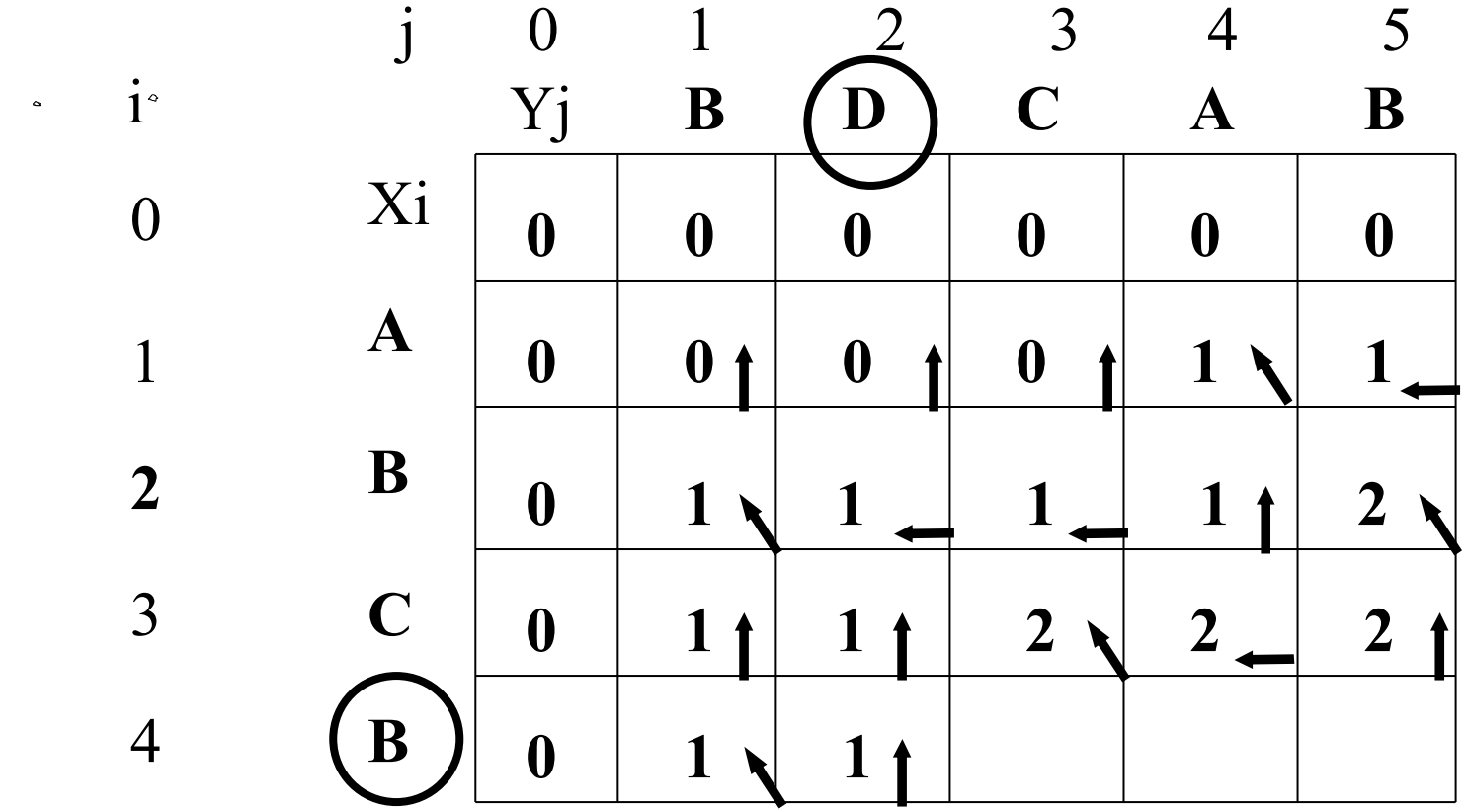


case i=4 and j=1

B = B

→ c[4, 1] = c[3, 0]+1 si b[4, 1] = ↖

Exemplu LCS (21)



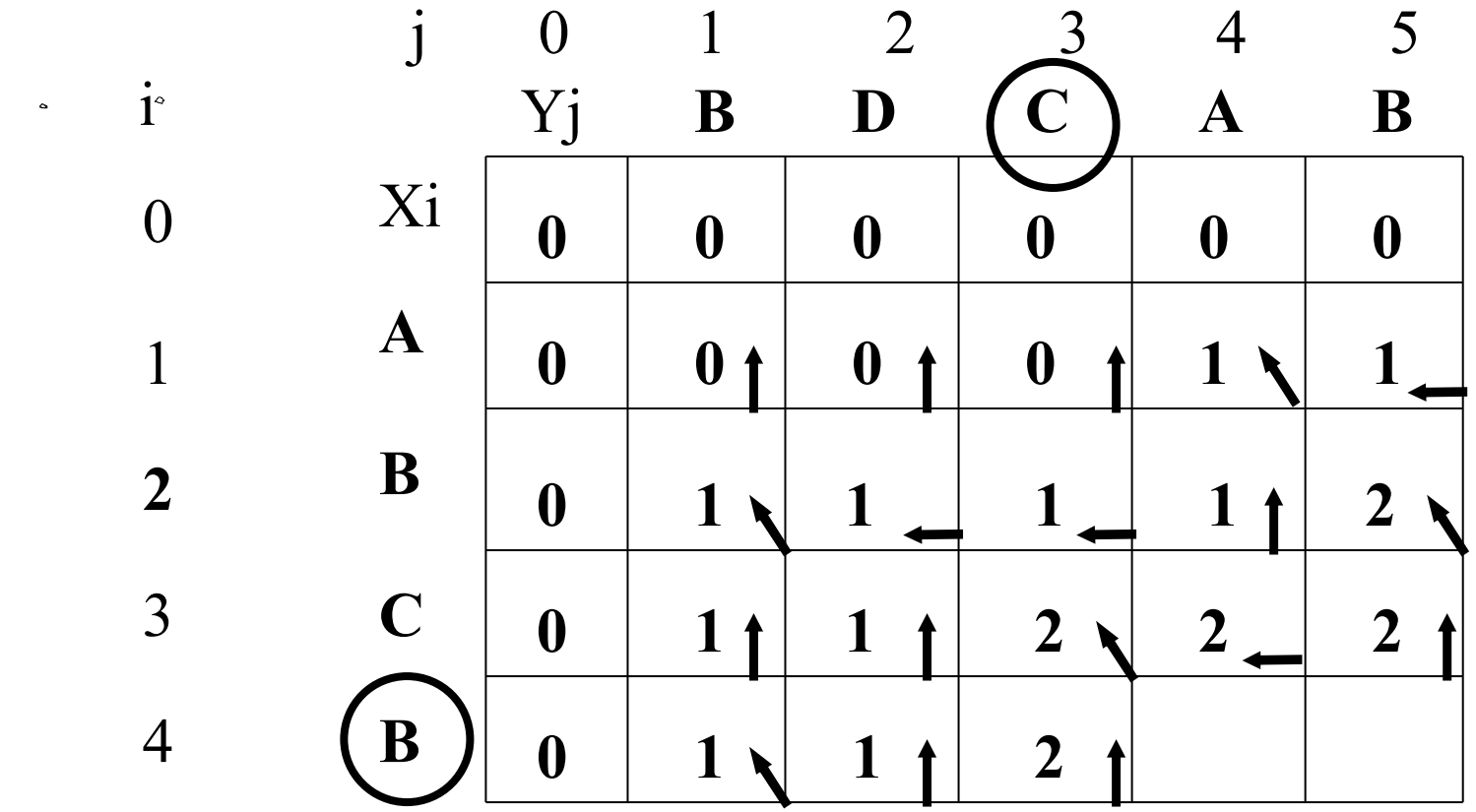
case i=4 and j=2

B != D

c[3, 2] = c[4, 1]

→ c[4, 2] = c[3, 2] si b[4, 2] = ↑

Exemplu LCS (22)



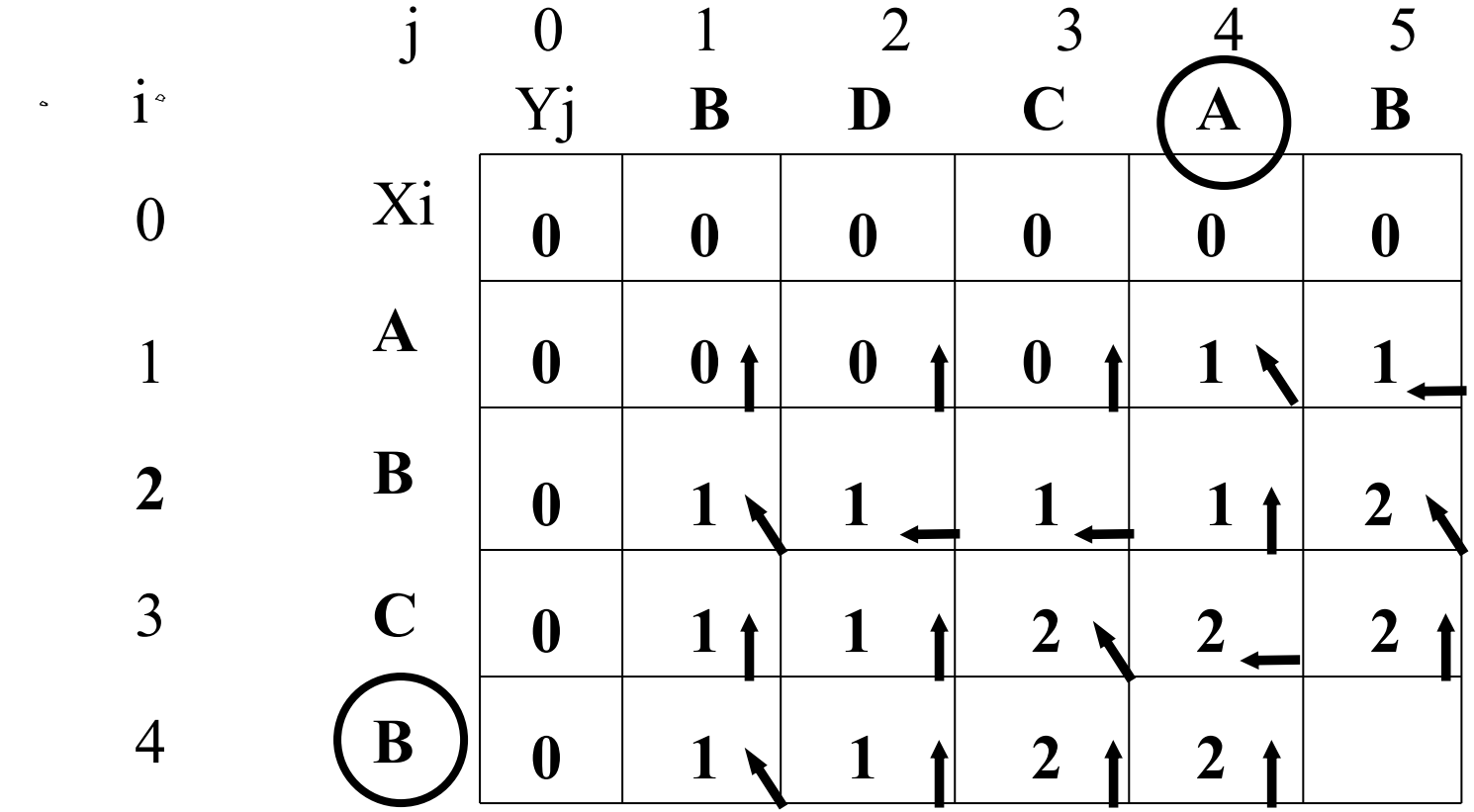
case i=4 and j= 3

B != C

c[3, 3] > c[4, 2]

→ c[4, 3] = c[3, 3] si b[4, 3] = ↑

Exemplu LCS (23)



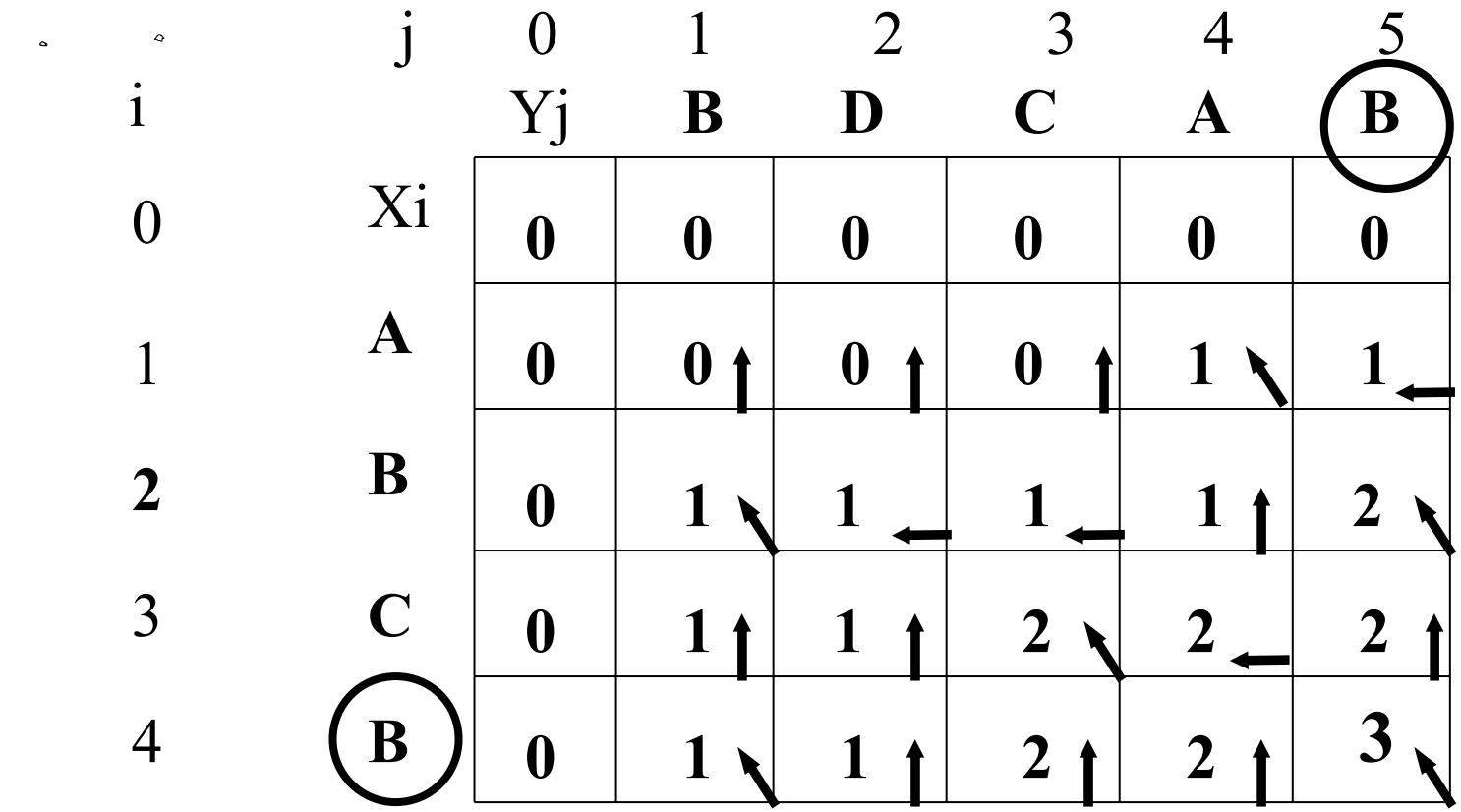
case i=4 and j=4

B != A

c[3, 4] = c[4, 3]

→ c[4, 4] = c[3, 4] si b[3, 5] = ↑

Exemplu LCS (24)



case i=4 and j=5

B= B

→ c[4, 5] = c[3, 4]+1 si b[4, 5] =

Algoritmul LCS

LCS-Length(X, Y)

m = length(X), n = length(Y)

for i = 1 to m do

c[i, 0] = 0

for j = 0 to n do

c[0, j] = 0

for i = 1 to m do

for j = 1 to n do

if (x_i = = y_j) then

c[i, j] = c[i - 1, j - 1] + 1

$b[i, j] = "\leftarrow\uparrow"$

else if c[i - 1, j]>=c[i, j - 1] then

c[i, j] = c[i - 1, j]

$b[i, j] = "\uparrow"$

else c[i, j] = c[i, j - 1]

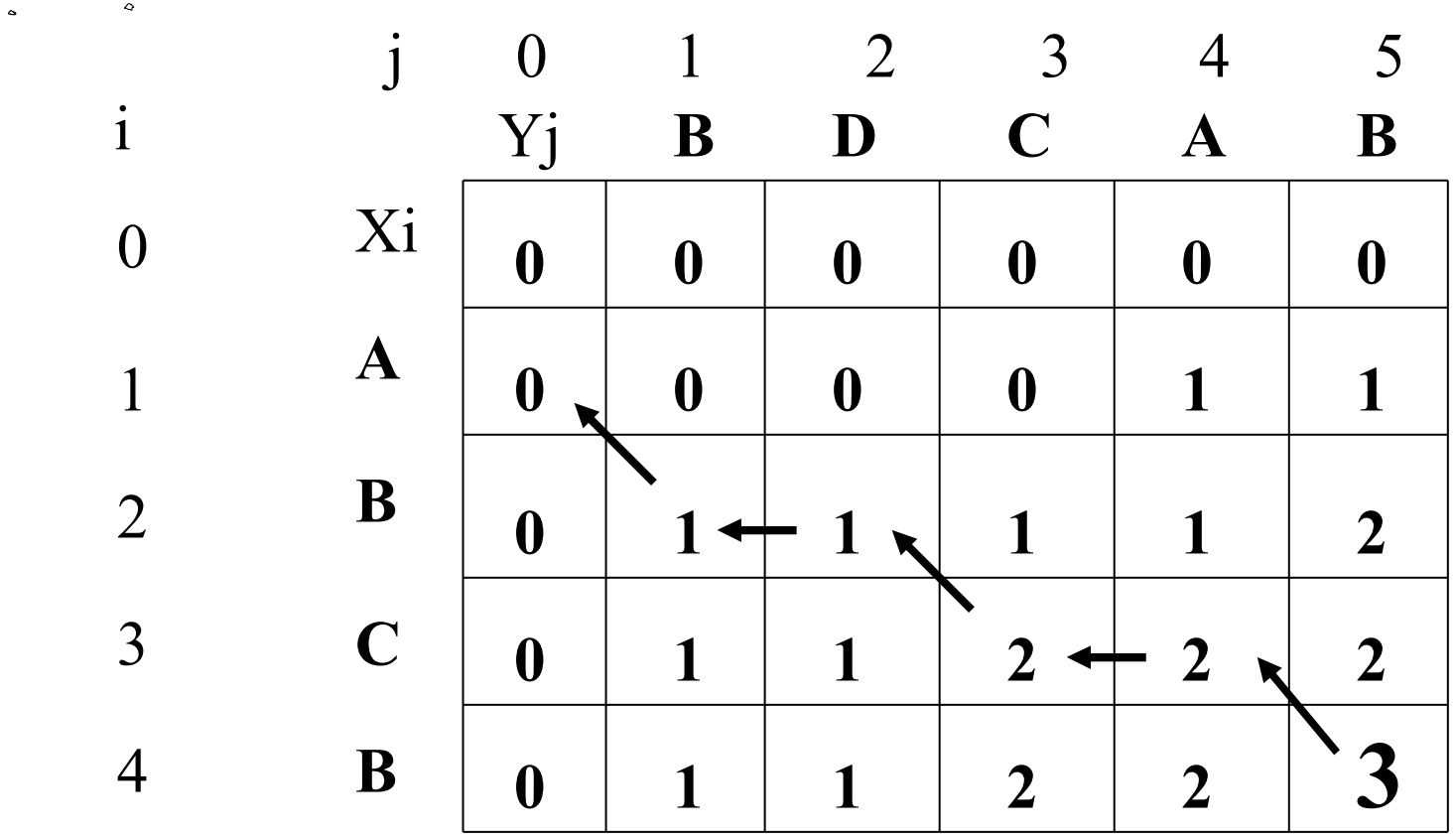
$b[i, j] = "\leftarrow"$

return c and b

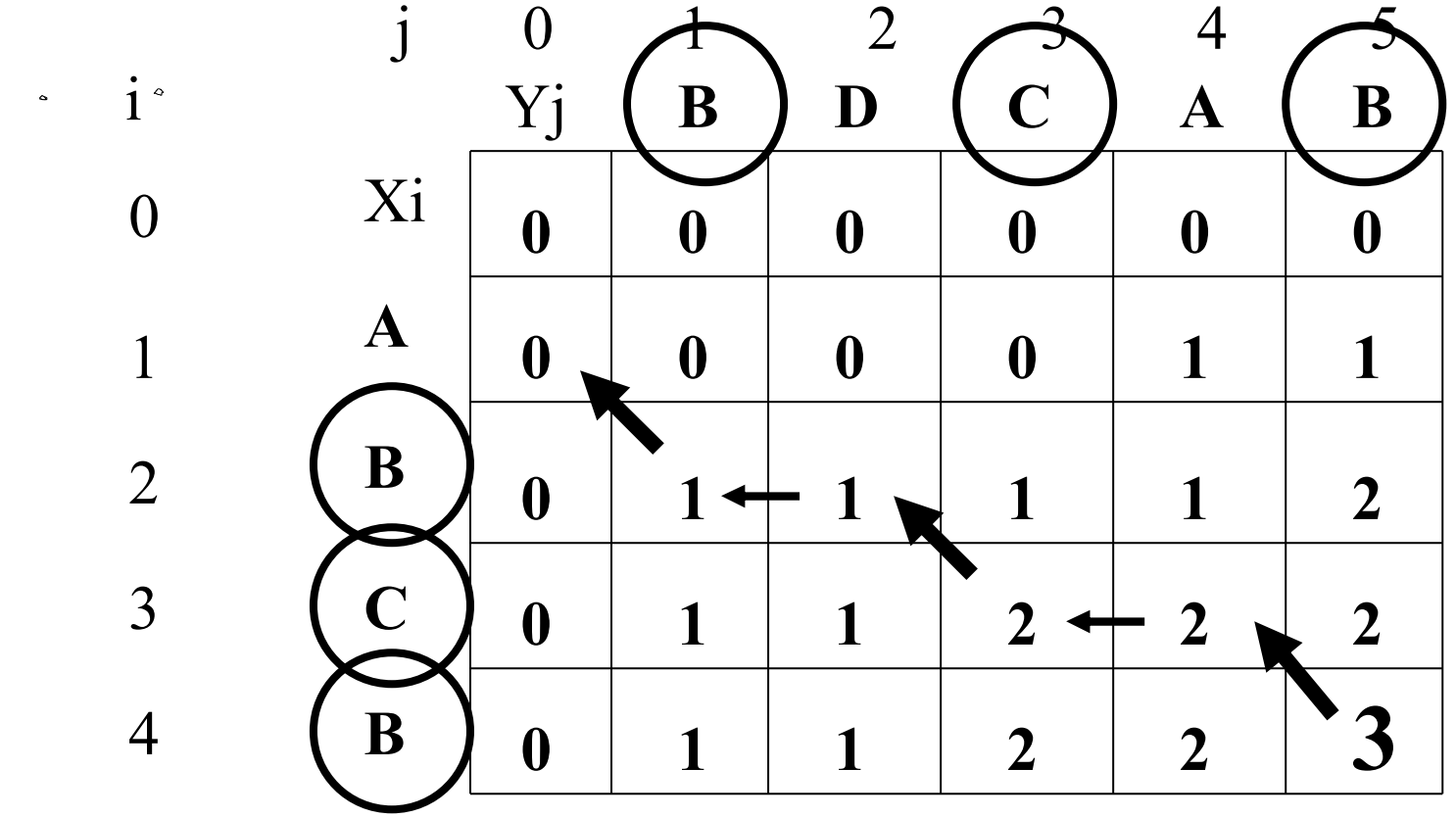
Cum lucreaza algoritmul LCS

- Algoritmul LCS algorithm calculeaza valorile pentru fiecare intrare a matricii c[m,n]
- Timp de lucru: O(mn)
- Cum poate fi gasita solutia urmatoare? Sagetile introduse reprezinta “un ghid” in acest scop. Sagetile sunt urmarite pentru a ajunge “inapoi” la *base case* 0

Cum poate fi gasit LCS (1)



Cum poate fi gasit LCS (2)



LCS: **B C B**

Cum poate fi gasit LCS (3)

- Print_LCS (X, i, j)
  - if i = 0 or j = 0 then return
  - if b[i, j] = “ ↖ “ then
    - Print_LCS (X, i-1, j-1)
    - Print X[i]
  - elseif b[i, j] = “ ↑ “ then
    - Print_LCS (X, i-1, j)
  - else
    - Print_LCS (X, i, j-1)

Cost: O(m+n)

Backtracking

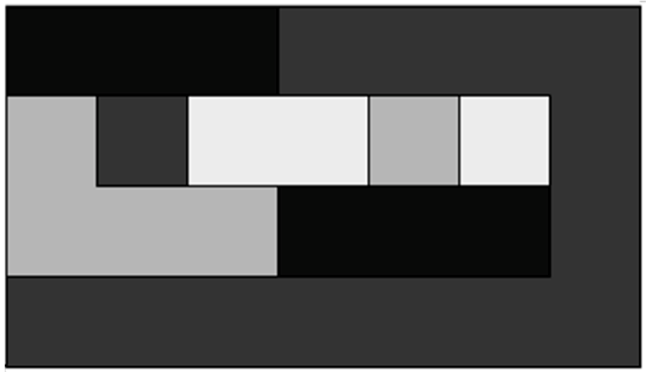
- Sa presupunem ca trebuiesc luate o serie de *decizii*, in raport cu mai multe *optiuni*, in urmatoarele conditii:
  - Nu exista suficiente informatii pentru a cunoaste exact ce solutie trebuie aleasa
  - Fiecare decizie conduce la un nou set de optiuni
  - Orice secventa de optiuni poate fi o solutie a problemei.
- Backtracking reprezinta o solutie metodica de a verifica diferite secvente de optiuni in scopul gasirii uneia functionale

Solutia labirintului

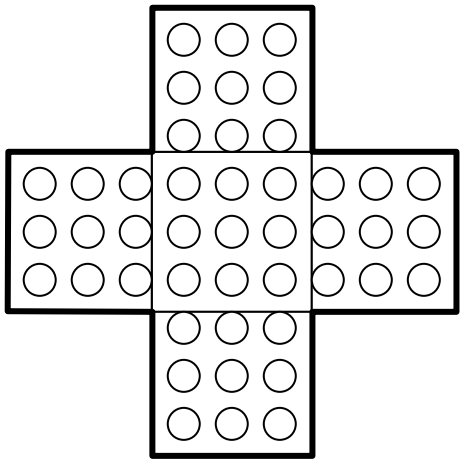
- Fiind dat un labirint, sa se gaseasca drumul de la intrare la iesire
- La fiece intersectie trebuie luata o decizie in raport cu (max) trei variante:
  - inainte
  - stanga
  - dreapta
- Nu exista suficiente informatii pentru alegerea corecta
- Fiecare optiune conduce la un alt set de optiuni
- Una sau mai multe secvente de optiuni poate / nu poate conduce la o solutie
- Algoritmul backtracking poate oferi o rezolvare pentru aceasta problema

Alte probleme...

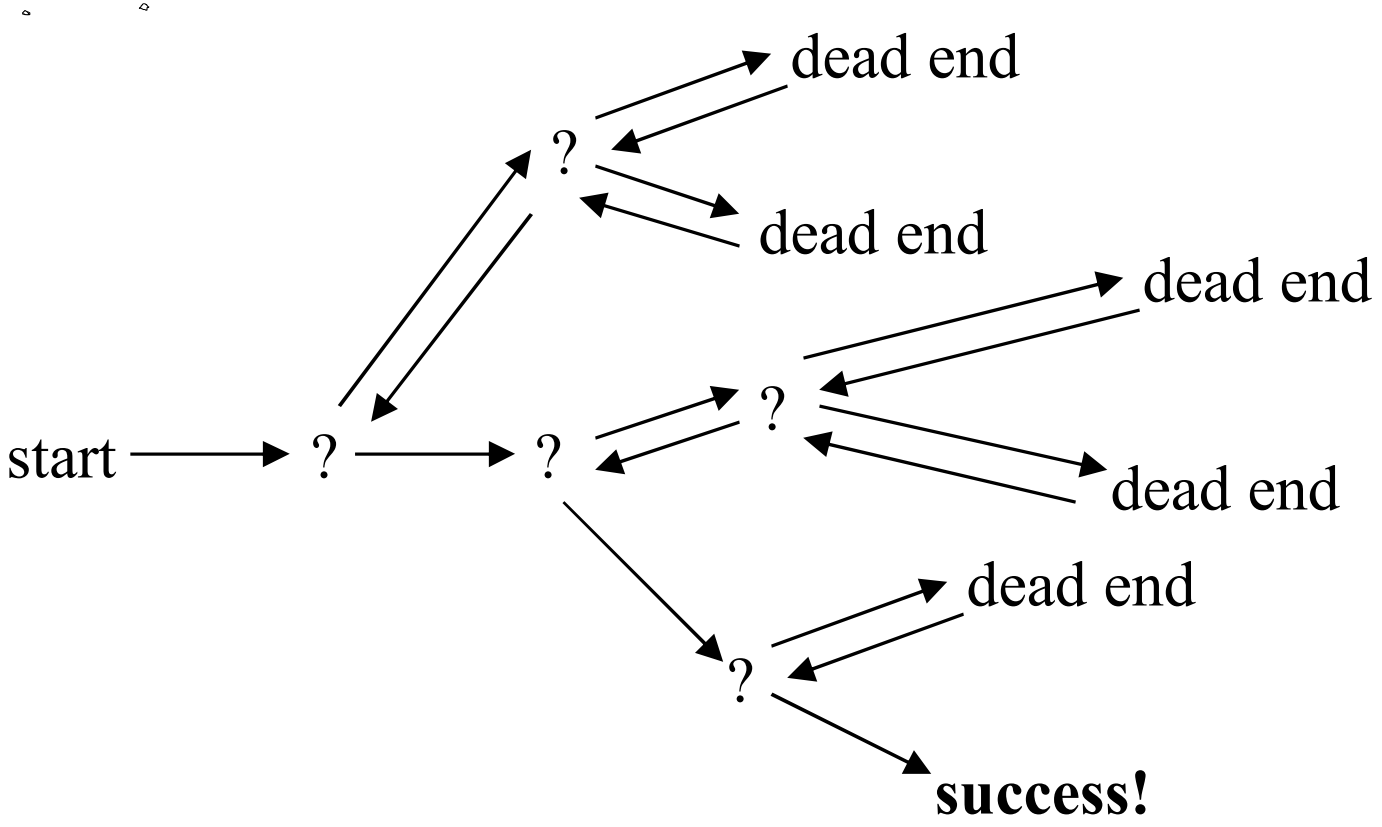
- Colorarea unei harti cu 4 culori:
  - Rosu, galben, verde, albastru
- Tarile adiacente trebuie sa fie in culori diferite



- In acest *puzzle*, toate gaurile (cu exceptia uneia) sunt umplute cu pioni albi
- Se poate sari cu un pion peste altul
- Pionii peste care s'a sarit ies din joc
- Scop: eliminarea tuturor pionilor (cu exceptia unuia)



Backtracking (parcurgere)



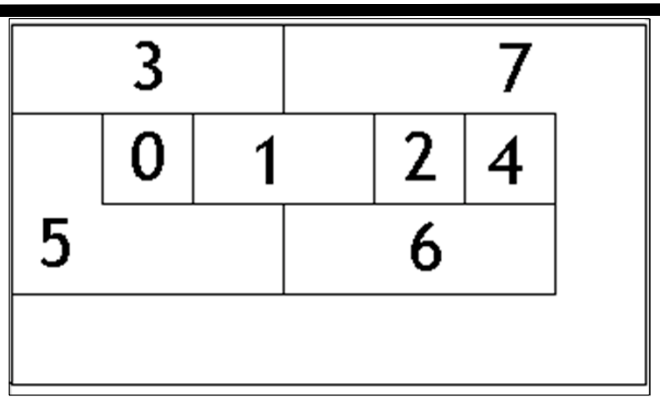
Algoritmul backtracking

- Implementare: se exploreaza fiecare nod (al unui arbore) in felul urmator :
- Pentru nodul N:
  1. If N is a goal node, return “success”
  2. If N is a leaf node, return “failure”
  3. For each child C of N,
     - 3.1. Explore C
       - 3.1.1. If C was successful, return “success”
  4. Return “failure”

Problema hartii – structura de date

- Structura de date trebuie sa permita:
  - Setarea unei culori pentru fiecare tara
  - Stabilirea vecinilor pentru fiecare tara
- Variante
  - Stabilirea unei matrici a culorilor: countryColor[i] este culoarea pentru tara *i*
  - O matrice pentru definirea vecinilor (tarilor adiacente) map[i][j] semnifica faptul ca tara *j* este adiacenta tarii *i*
    - Exemplu: map[5][3]==8 → a treia tara adiacenta tarii 5 este tara 8

Crearea hartii




```
int map[][];

void createMap() {
    map = new int[][] { { 1, 3, 5 },           // adj to 0
                        { 0, 2, 3, 5, 6, 7 },  // adj to 1
                        { 1, 4, 6, 7 },        // adj to 2
                        { 0, 1, 5, 7 },        // adj to 3
                        { 2, 6, 7 },           // adj to 4
                        { 0, 1, 3, 6, 7 },     // adj to 5
                        { 1, 2, 4, 5, 7 },     // adj to 6
                        { 1, 2, 3, 4, 5, 6 } }; // adj to 7
}
```

Exemplu preluat dupa *Recursive Back Tracking & Dynamic Programming,* Jeff Edmonds, York University

Setare culori initiale

```
static final int NONE = 0;
static final int RED = 1;
static final int YELLOW = 2;
static final int GREEN = 3;
static final int BLUE = 4;

int mapColors[] = { NONE, NONE, NONE, NONE,
                    NONE, NONE, NONE, NONE };
```

Programul principal

- (Clasa: ColoredMap)

```
public static void main(String args[]) {
    ColoredMap m = new ColoredMap();
    m.createMap();
    boolean result = m.explore(0, RED);
    System.out.println("Solutie = " + result);
    m.printMap();
}
```

Metoda backtracking

```
boolean explore(int country, int color) {
    if (country >= map.length) return true;
    if (okToColor(country, color)) {
        mapColors[country] = color;
        for (int i = RED; i <= BLUE; i++) {
            if (explore(country + 1, i)) return true;
        }
    }
    mapColors[country] = NONE;
    return false;
}
```

Verificare utilizare culoare

```
boolean okToColor(int country, int color) {
    for (int i = 0; i < map[country].length; i++) {
        int ithAdjCountry = map[country][i];
        if (mapColors[ithAdjCountry] == color) {
            return false;
        }
    }
    return true;
}
```

Solutie

- Solutie gasita = true
  - map[0] is red
  - map[1] is yellow
  - map[2] is green
  - map[3] is blue
  - map[4] is yellow
  - map[5] is green
  - map[6] is blue
  - map[7] is red