NP-Completitudine

Algoritmi exponenţiali

Metode de elaborare a algoritmilor

- **Greedy –** probleme de optim

  - este necesară demonstrarea corectitudinii

  - algoritmi polinomiali

- **Divide et impera**

- **Programare dinamică**

Metode de elaborare a algoritmilor

- **Greedy –** probleme de optim

  - corectitudine (v. curs moodle actualizat)

  - algoritmi polinomiali

- **Divide et impera –** subprobleme de acelaşi tip

  - **construirea dinamică a unui arbore** (prin împărţirea în subprobleme) urmată de **parcurgerea în postordine a arborelui** (prin asamblarea rezultatelor parţiale).

  - algoritmi polinomiali

- **Programare dinamică**

Metode de elaborare a algoritmilor

- **Greedy –** probleme de optim

  - corectitudine (v. curs moodle actualizat)

  - algoritmi polinomiali

- **Divide et impera –** subprobleme de acelaşi tip

  - **construirea dinamică a unui arbore**

  (prin împărţirea în subprobleme) urmată de **parcurgerea în postordine a arborelui** (prin asamblarea rezultatelor parţiale).

  - algoritmi polinomiali

- **Programare dinamică –** rezolvare de recurenţe->PD-arbore

  - principiu de optimalitate

  - **parcurgerea în “postordine” generalizată a PD-arborelui**

  - algoritmi polinomiali + **pseudo** polinomiali

Metode de elaborare a algoritmilor

- Complexitatea în timp a algoritmilor joacă un rol esenţial.
- **Un algoritm este considerat "acceptabil" numai dacă timpul său de executare este polinomial**

Metode de elaborare a algoritmilor

Nu ştim algoritm polinomial – problemă grea?

Metode de elaborare a algoritmilor

Nu ştim algoritm polinomial – problemă grea?

P = clasa problemelor pentru care există algoritmi polinomiali (determinişti)

Metode de elaborare a algoritmilor

Nu ştim algoritm polinomial – problemă grea?

NP

- există algoritm polinomial pentru a testa o soluţie candidat dacă este soluţie posibilă (verificator polinomial)

⇒ o problemă NP poate fi rezolvată în timp exponenţial (considerând toate soluţiile candidat)

Metode de elaborare a algoritmilor

Nu ştim algoritm polinomial – problemă grea?

NP

- P ≠ NP ?

- Probleme NP-complete (NP, NP-hard)

- B ∈ NP a.î. ∀A ∈ NP, A ≤ p B.
- Dacă pentru una se găsește algoritm polinomial, atunci P = NP
- SAT

- Probleme NP-dificile (NP-hard)

- B a.î. ∀A ∈ NP, A ≤ p B.

Metode de elaborare a algoritmilor

Nu ştim algoritm polinomial

- **Demonstrăm** NP – dificilă

Soluţii:

- algoritmi exponenţiali mai rapizi decât cei exhaustivi (brute force) de căutare în spaţiul soluţiilor: **Backtracking, Branch & Bound**

- **Compromis**: algoritmi mai rapizi care produc soluţii care nu sunt optime – algoritmi **euristici, aleatorii, genetici**…

- O euristică este o metodă de a clasifica alegerile posibile la un anumit pas (cu scopul de a determina o alegere optimă a pasului următor) în explorarea spațiului soluțiilor unei probleme.

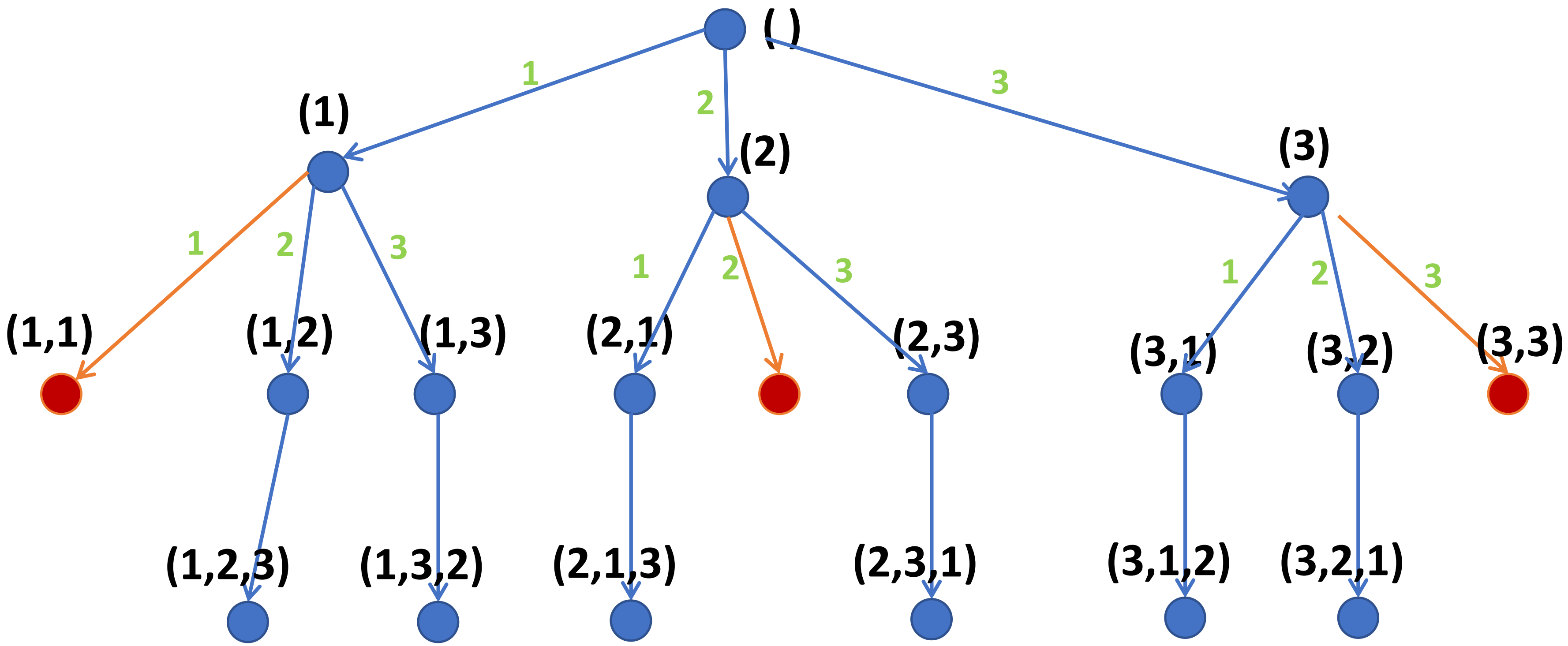
Backtracking,
Branch and Bound

Cadru comun

- Căutare mai "inteligentă" în spaţiul în care se găsesc soluţiile posibile, reprezentate de obicei cu ajutorul vectorilor
- Configuraţiile prin care se trece în procesul de căutare - structură arborescentă

Cadru comun

- Structură arborescentă – permutări `{1, 2, 3}`



Cadru comun

- Parcurgerea completă a arborelui ⇒ algoritm exhaustiv (brute force), care consideră toate soluţiile candidat

Cadru comun

- Parcurgerea completă a arborelui ⇒ algoritm exhaustiv (brute force), care consideră toate soluţiile candidat
- Mai rapid – **limitarea parcurgerii** arborelui prin determinarea de configuraţii care **nu pot conduce către soluţii** dorite, care nu mai sunt explorate

Cadru comun

- Parcurgerea completă a arborelui ⇒ algoritm exhaustiv (brute force), care consideră toate soluţiile candidat
- Mai rapid – **limitarea parcurgerii** arborelui prin determinarea de configuraţii care **nu pot conduce către soluţii** dorite, care nu mai sunt explorate
- **Diferenţe**

– modul în care este parcurs arborele (şi în care se fac limitările)

= criteriul după care este ales nodul curent

- tipuri de probleme la care se pretează

Metoda Backtracking

Cadru

- $X = X_1 \times \ldots \times X_n$ = **spaţiul soluţiilor posibile**
- $\varphi : X \to \{0,1\}$ este o **proprietate (constrângere)** definită pe X
- **Căutăm un vector $x \in X$ cu proprietatea $\varphi(x)$**
    - **condiții interne** pentru x

Cadru

- $X = X_1 \times \ldots \times X_n$ = **spaţiul soluţiilor posibile**
- $\varphi : X \to \{0,1\}$ este o **proprietate (constrângere)** definită pe X
- **Căutăm un vector $x \in X$ cu proprietatea $\varphi(x)$**
    - <u>**condiții interne**</u> pentru x
- Vectorul soluție $x = (x_1, x_2, \ldots, x_n) \in X$ este **construit progresiv**, începând cu prima componentă. Componenta i va fi completată cu o valoare din domeniul X_i.
- Metoda backtracking încearcă micşorarea timpului de calcul - prin **evitarea generării unor soluţii care nu satisfac condiţiile interne**

Metoda Backtracking

- Configurațiile corespunzătoare soluțiilor parțiale ("incomplete"), pe care le putem **testa** dacă pot fi completate până la o soluție posibilă -> condiții de continuare

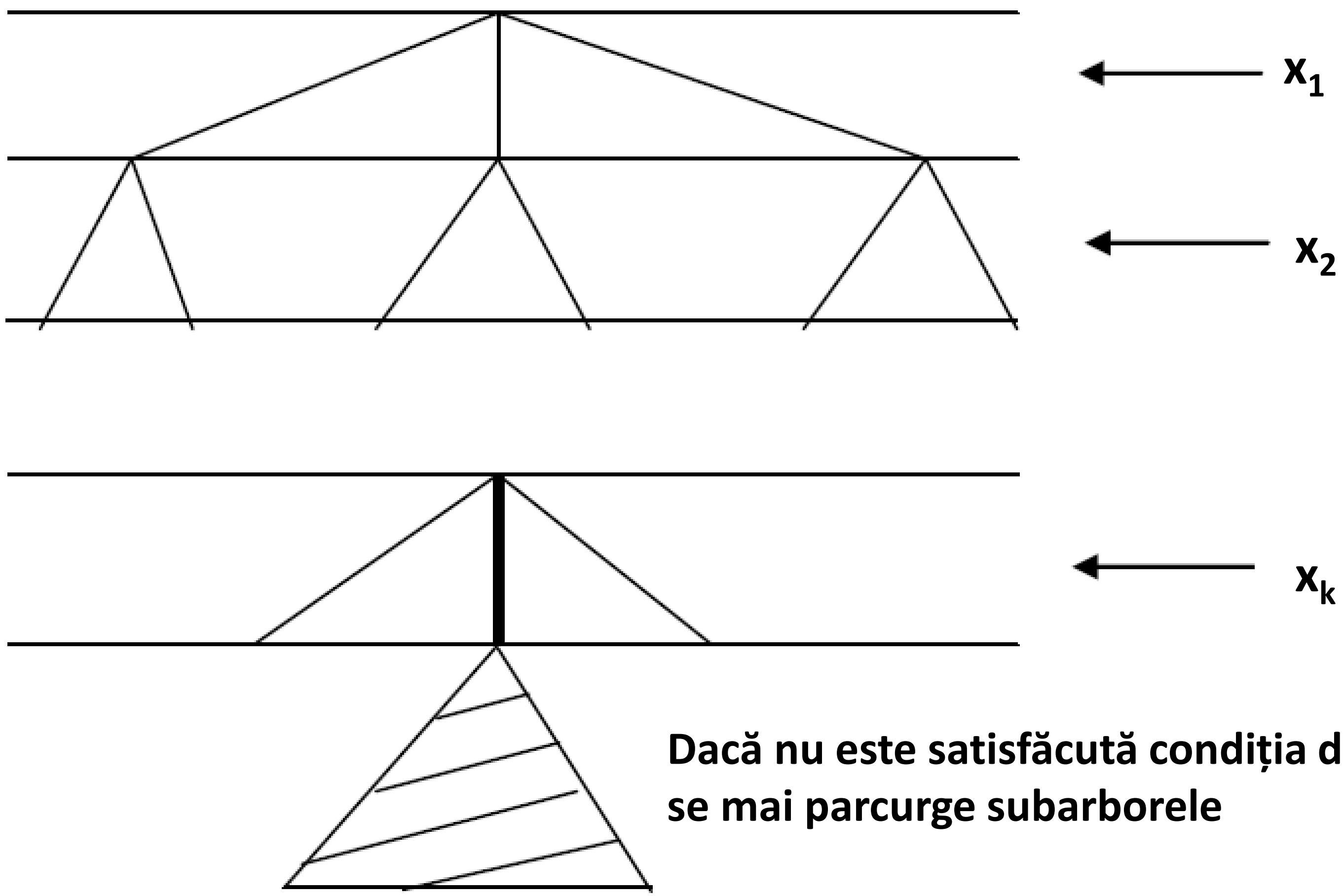
  - **condiții de continuare** pentru soluția parțială $y=x_1...x_k$ notate $cont_k(x)$ = condiții de continuare a parcurgerii subarborelui de rădăcină x

  - Se avansează cu o valoare pentru x_k dacă este satisfăcută **condiţia de continuare** $cont_k(x_1,...,x_k)$.

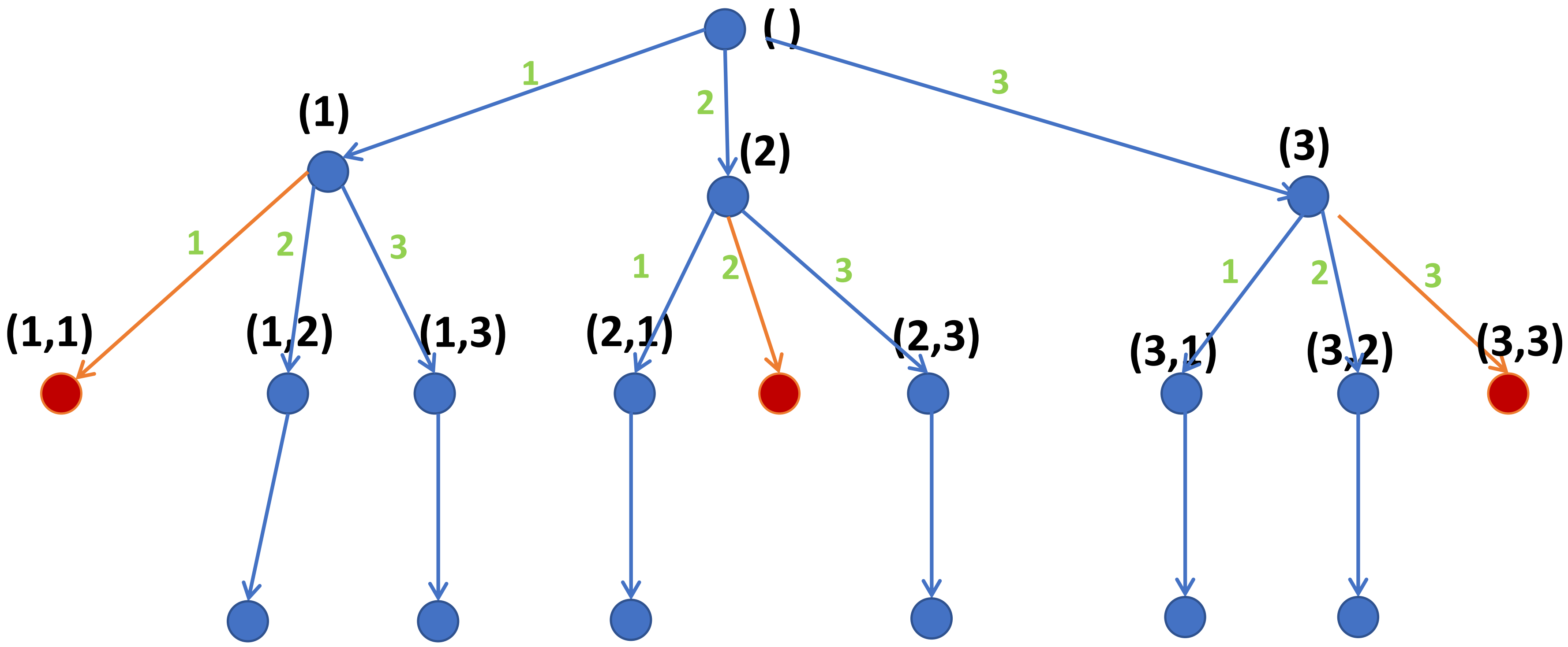
  - **Condiţiile de continuare rezultă de obicei din φ.** Ele sunt **strict necesare**, **ideal fiind să fie şi suficiente**.

Metoda Backtracking

- **Backtracking = parcurgerea limitată (de condiţiile de continuare) în adâncime a unui arbore**



Dacă nu este satisfăcută condiția de continuare nu se mai parcurge subarborele

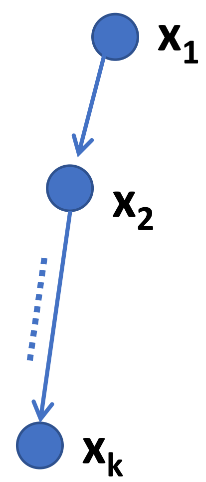

( )
1
2
3
(1)
(2)
(3)
1
2
3
1
2
3
1
2
3
(1,1)
(1,2)
(1,3)
(2,1)
(2,3)
(3,1)
(3,2)
(3,3)

Metoda Backtracking

- Cazuri posibile la alegerea lui x_k:
  - ❑ **Atribuie şi avansează**
  - ❑ **Încercare eşuată**
  - ❑ **Revenire**
  - ❑ **Revenire după determinarea unei soluţii**

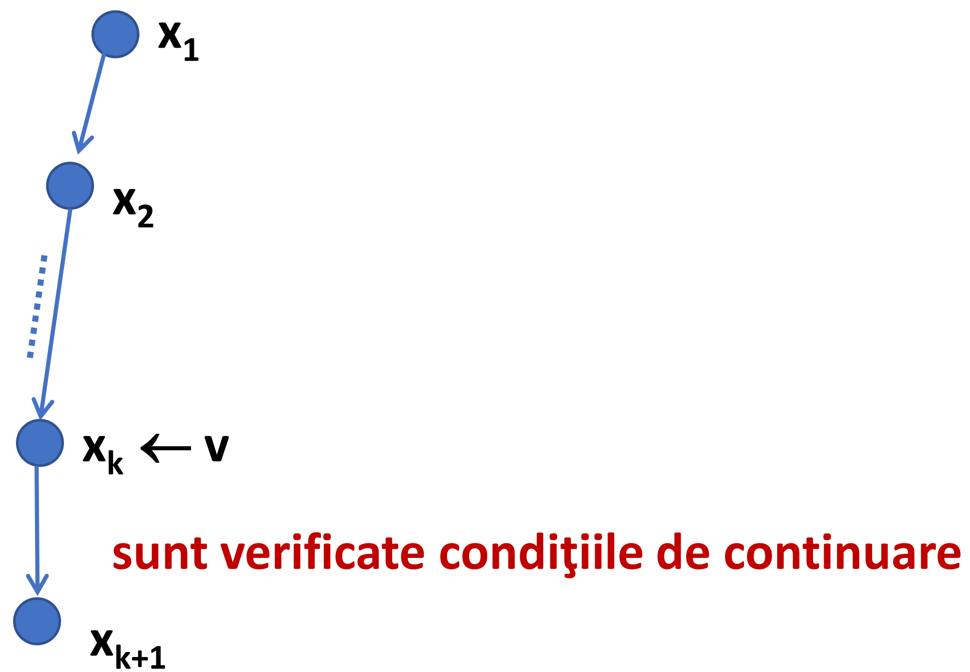
Metoda Backtracking

- Cazuri posibile la alegerea lui x_k:



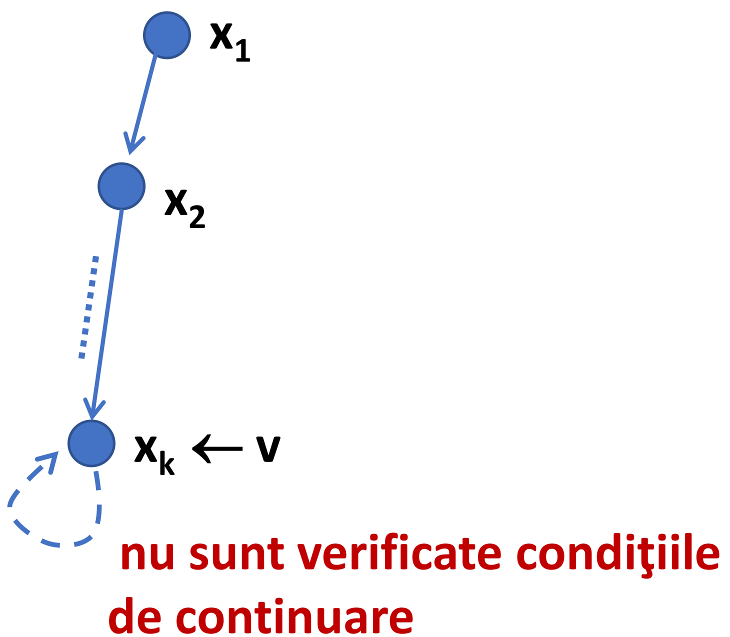
Metoda Backtracking

- **Atribuie o valoare $v \in X_k$ lui x_k şi avansează (sunt verificate condiţiile de continuare)**



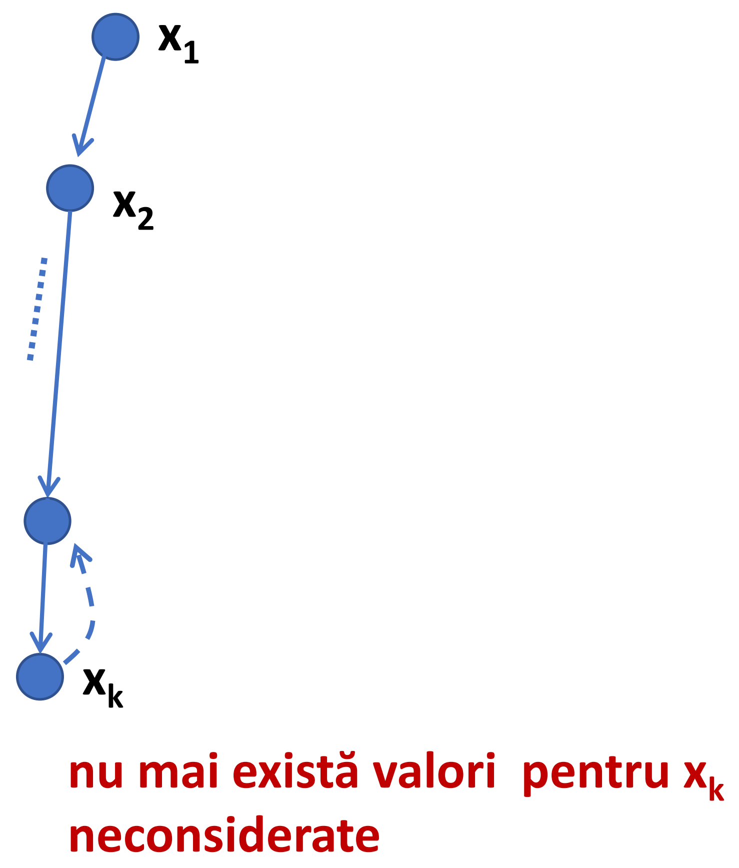
Metoda Backtracking

- **Încercare eşuată (atribuie o valoare $v \in X_k$ lui x_k pentru care nu sunt verificate condiţiile de continuare)**



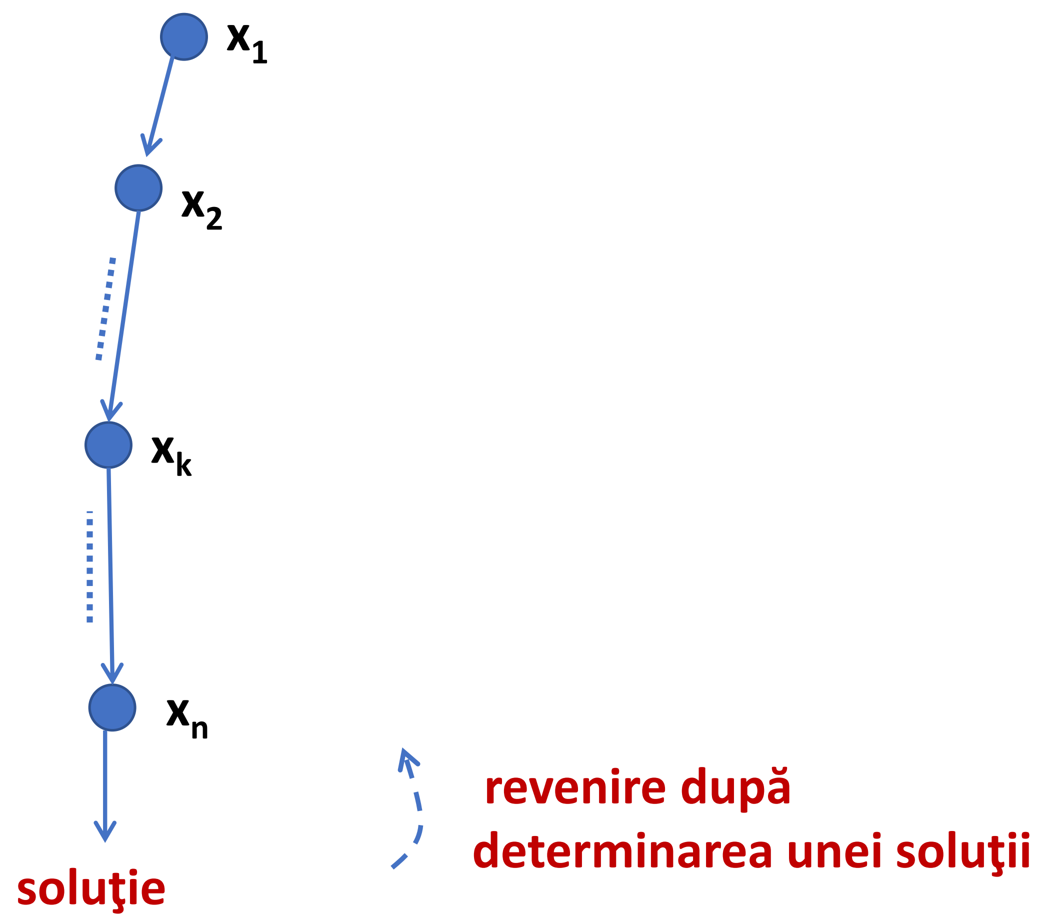
Metoda Backtracking

- **Revenire - nu mai există valori $v \in X_k$ neconsiderate**



Metoda Backtracking

❑ **Revenire după determinarea unei soluţii**



Varianta nerecursivă – pseudocod

– pentru soluţii cu lungime fixă-

Cazul $X_i = \{p_i, p_i+1, \ldots, u_i\}$

$x_i \leftarrow \mathbf{p_i - 1}$, $\forall i=1,...,n$

$k \leftarrow 1;$

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
  if k=n+1
      retsol(x); k←k-1;{revenire după o sol.}
  else
```

```
xᵢ←pᵢ-1, ∀i=1,...,n
k←1;
while k>0
  if k=n+1
      retsol(x); k←k-1;{revenire după o sol.}
  else
      if xₖ<uₖ  {mai sunt valori în Xₖ}
          xₖ←xₖ+1;
```

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
  if k=n+1
      retsol(x); k←k-1;{revenire după o sol.}
  else
      if xk<uk  {mai sunt valori în Xk}
         xk←xk+1;
         if cont(x1,…,xk)
              k←k+1;    { atribuie şi avansează }
         else              { încercare eşuată }
```

```
xᵢ←pᵢ-1, ∀i=1,...,n
k←1;
while k>0
  if k=n+1
      retsol(x); k←k-1;{revenire după o sol.}
  else
      if xₖ<uₖ  {mai sunt valori în Xₖ}
          xₖ←xₖ+1;
         if cont(x₁,…,xₖ)
             k←k+1;   { atribuie şi avansează }
         else             { încercare eşuată }
      else xₖ←pₖ-1; k←k-1;     { revenire }
```

Cazul general

• C_k = mulţimea valorilor consumate din X_k

```
Cᵢ←∅, ∀i;
k←1;
while k>0
 if k=n+1
   retsol(x); k←k-1;{revenire după o soluţie}
 else
    if Cₖ ≠ Xₖ
      alege v∈Xₖ\Cₖ; Cₖ←Cₖ∪{v};
      if cont(x₁,…,xₖ₋₁,v)
            xₖ←v; k←k+1; { atribuie şi avansează }
      else                    { încercare eşuată }
   else Cₖ←∅; k←k-1;          { revenire }
```

Varianta recursivă

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **back(1)**

```
procedure back(k)
 if k=n+1
     retsol(x) {revenire dupa solutie}
 else




end.
```

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **back(1)**

```
procedure back(k)
 if k=n+1
     retsol(x) {revenire dupa solutie}
 else
     for (i=pk; i<=uk; i++) {valori posibile}
          xk←i;                   {atribuie}




end.
```

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **back(1)**

```
procedure back(k)
 if k=n+1
     retsol(x) {revenire dupa solutie}
 else
     for (i=pk; i<=uk; i++) {valori posibile}
           xk←i;                    {atribuie}
           if cont(x1,…,xk)
            back(k+1);        {avanseaza}
            {revenire din recursivitate}
end.
```

Exemple

Exemple – de ştiut

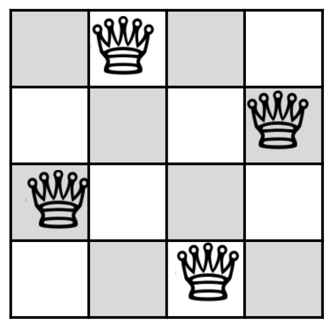
- **Permutări, combinări, aranjamente**
- **Colorarea hărților**
- **Problema ciclului hamiltonian**

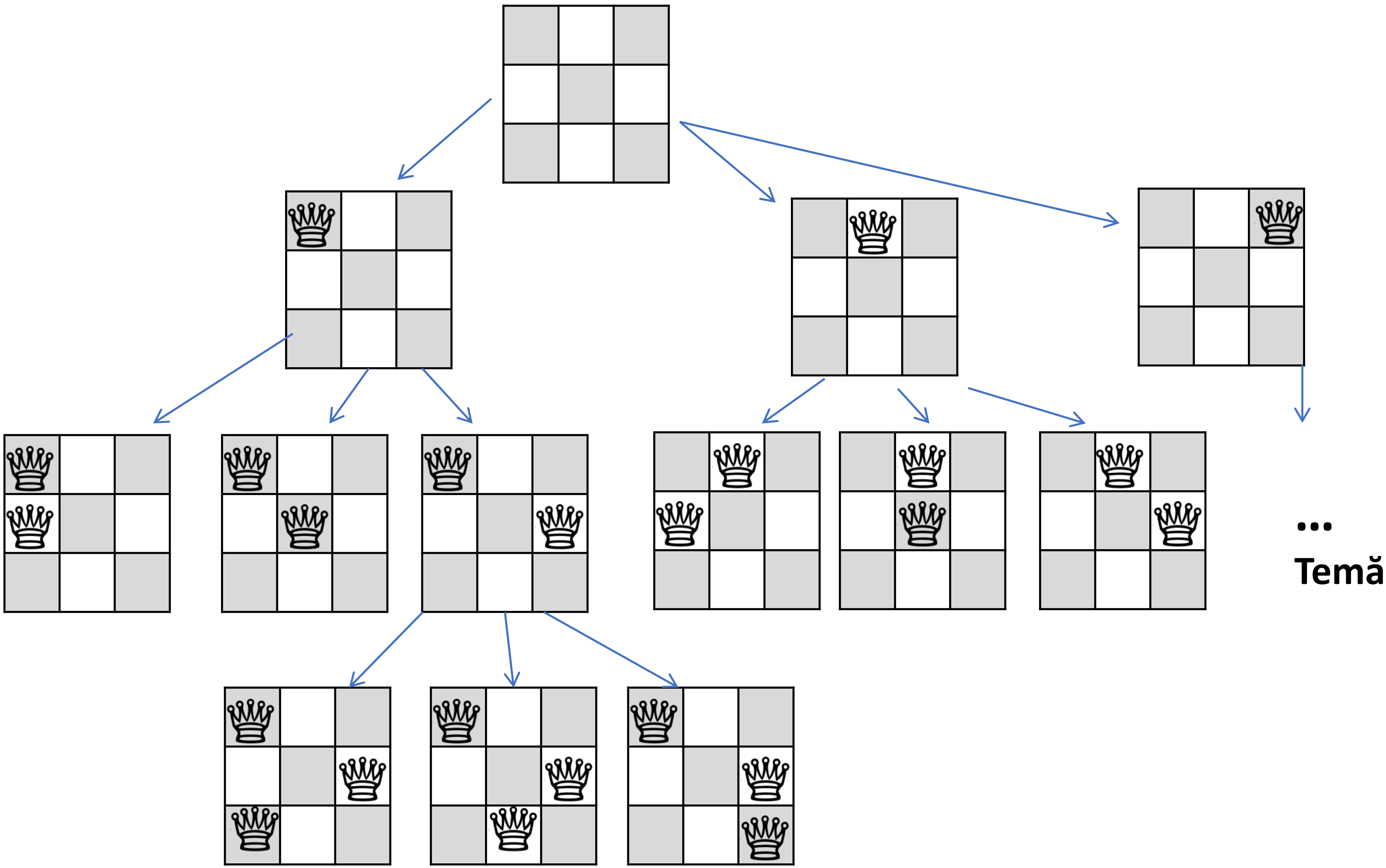
Pentru a testa condiţiile de continuare $\varphi_k(x_1, \ldots, x_k)$ vom folosi funcţia `cont(k)`

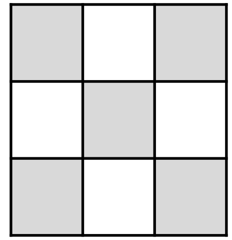
Problema celor n dame

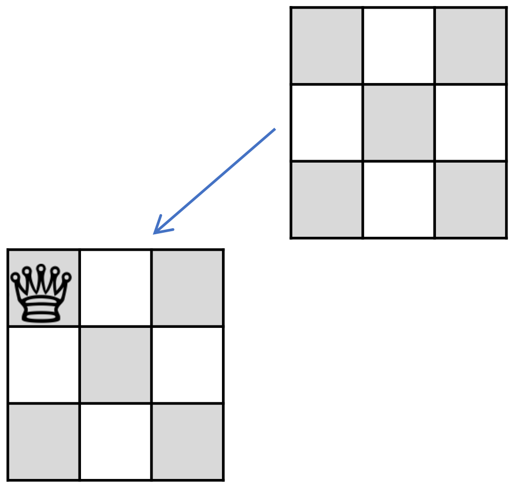
- Se consideră un caroiaj n×n.

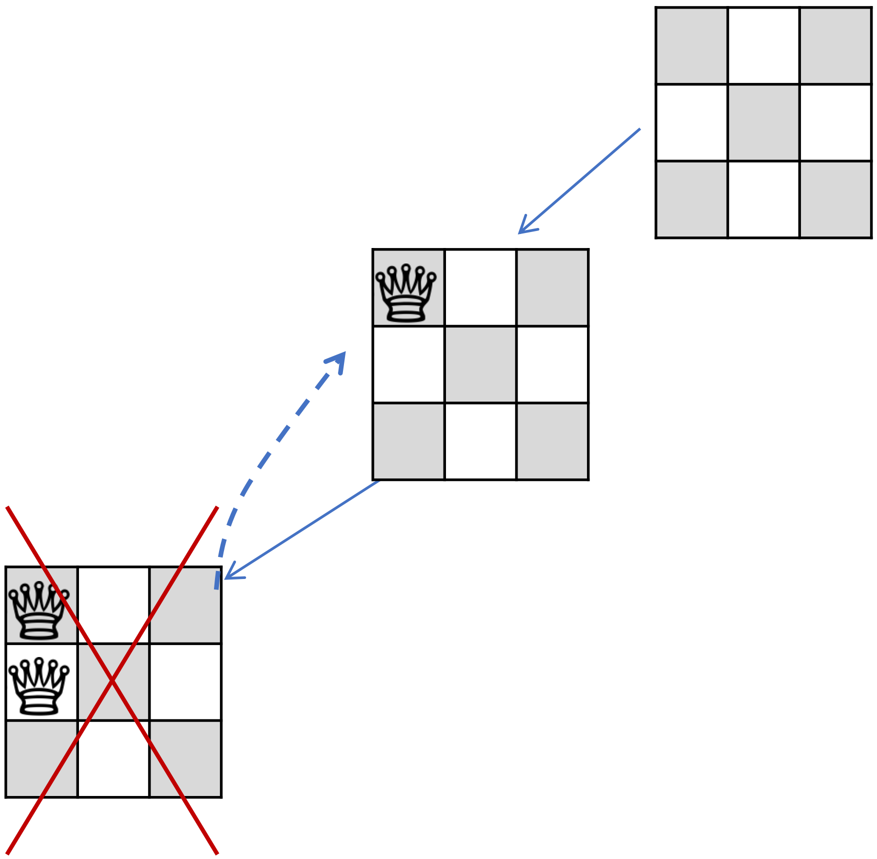
Prin analogie cu o tablă de şah (n=8), se doreşte plasarea a n dame pe pătrăţelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte

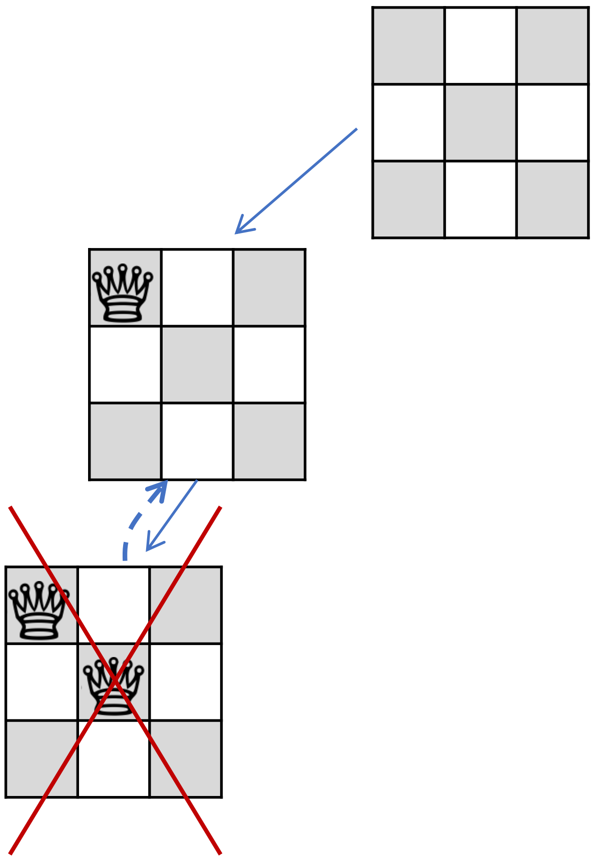


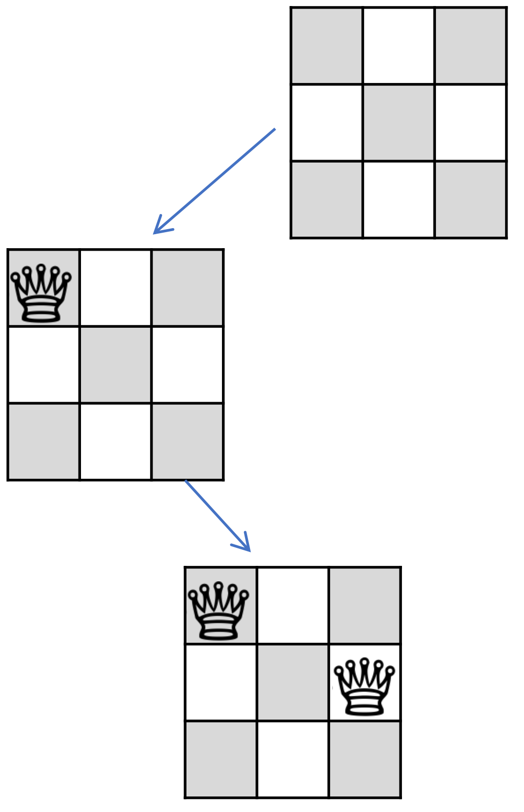

…
Temă

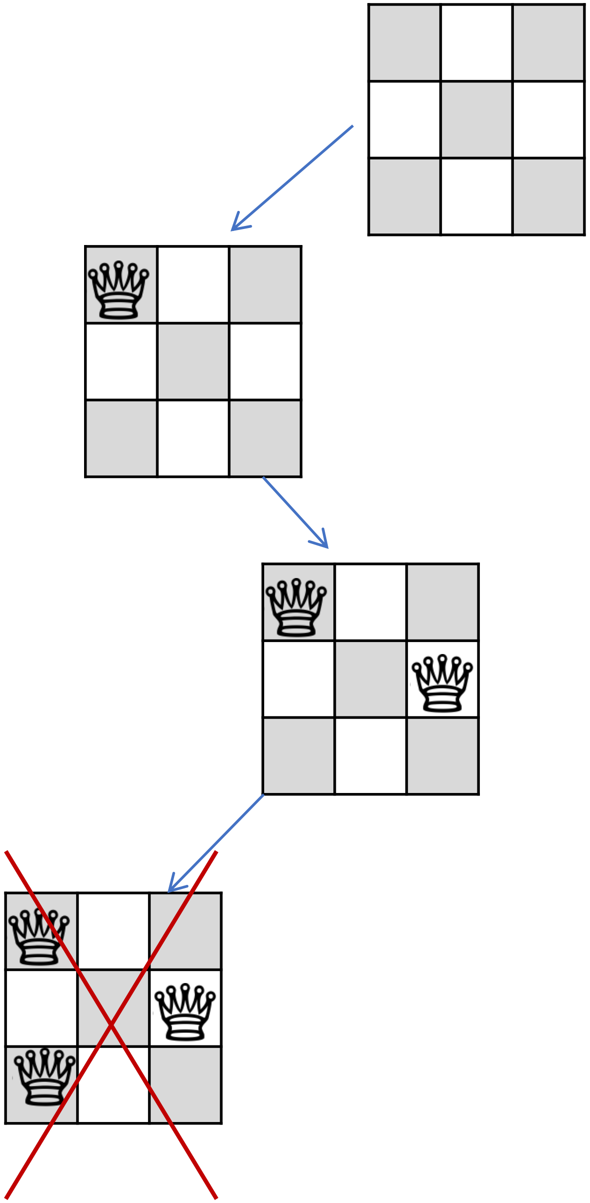


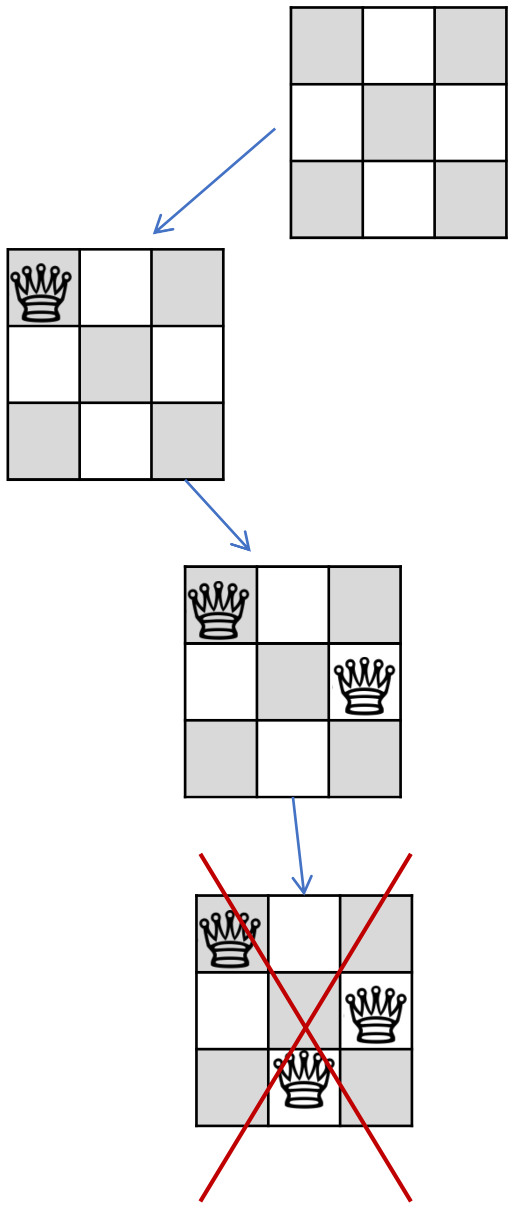


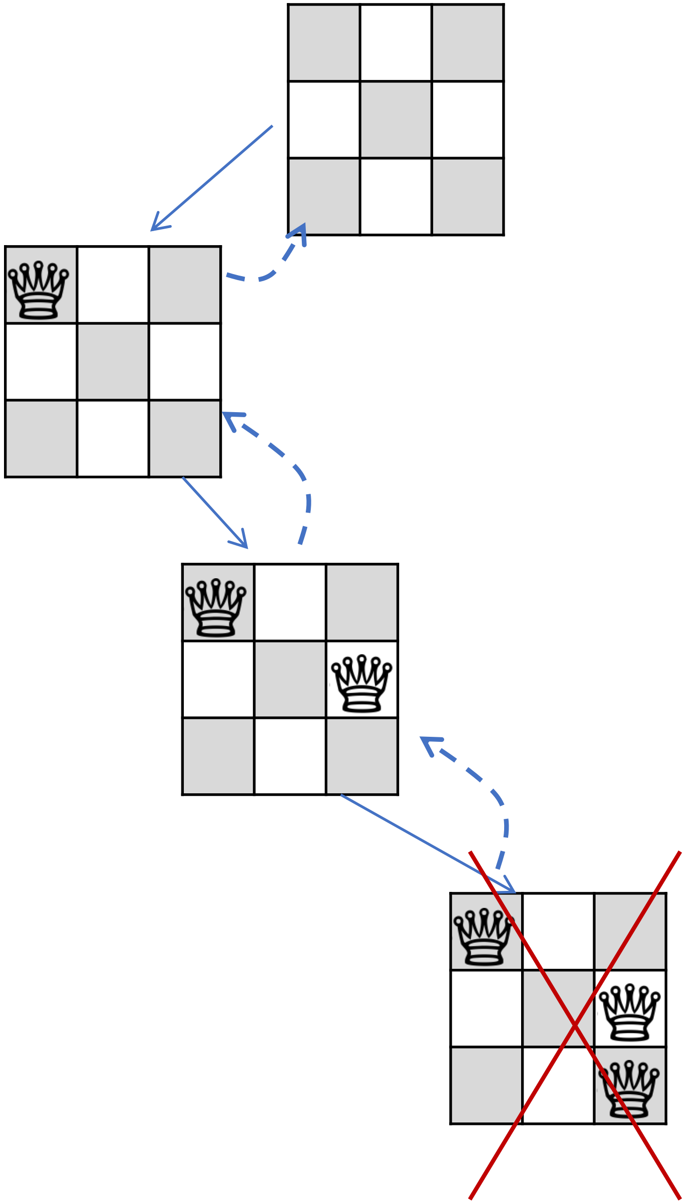


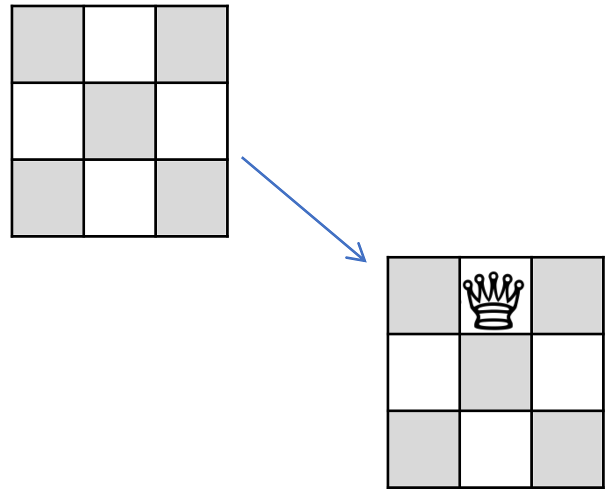


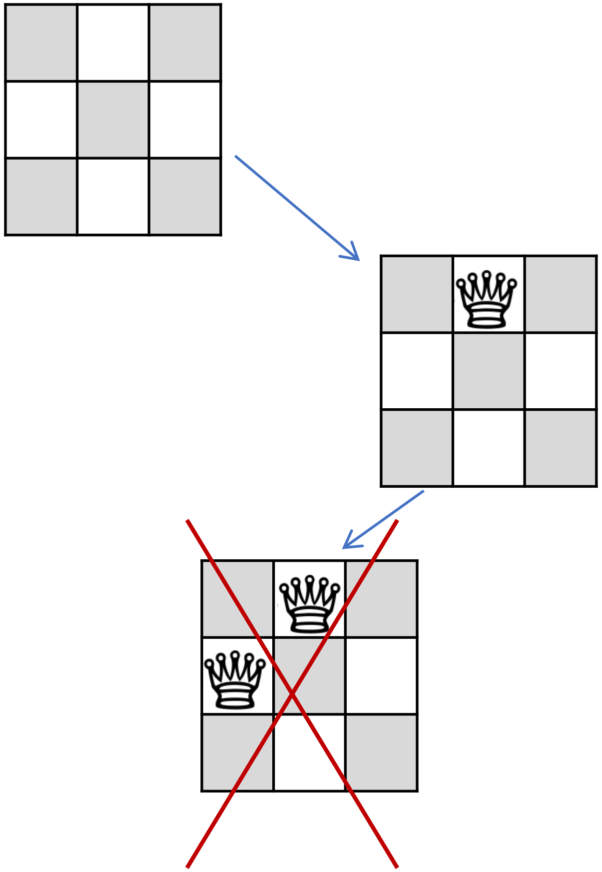


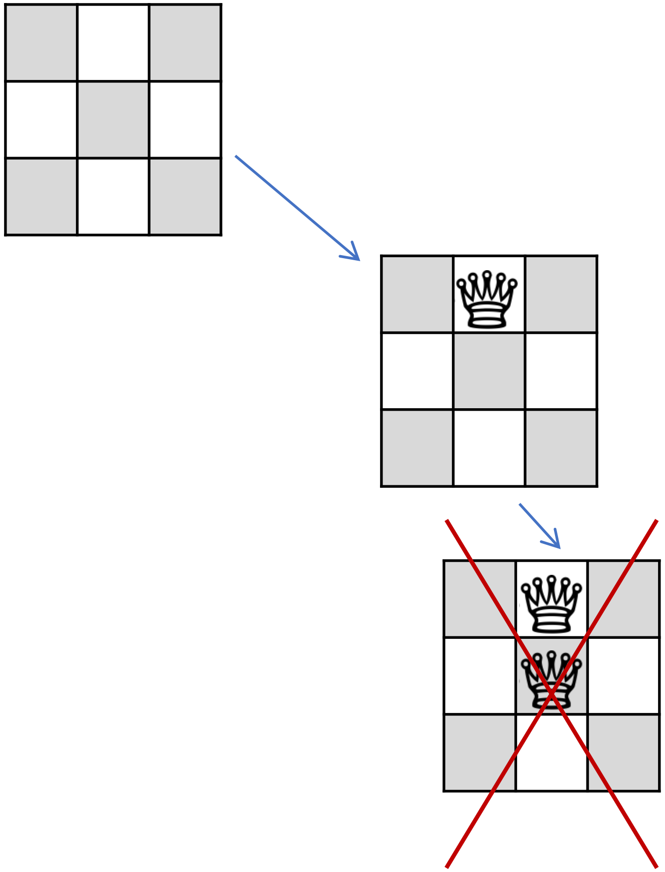


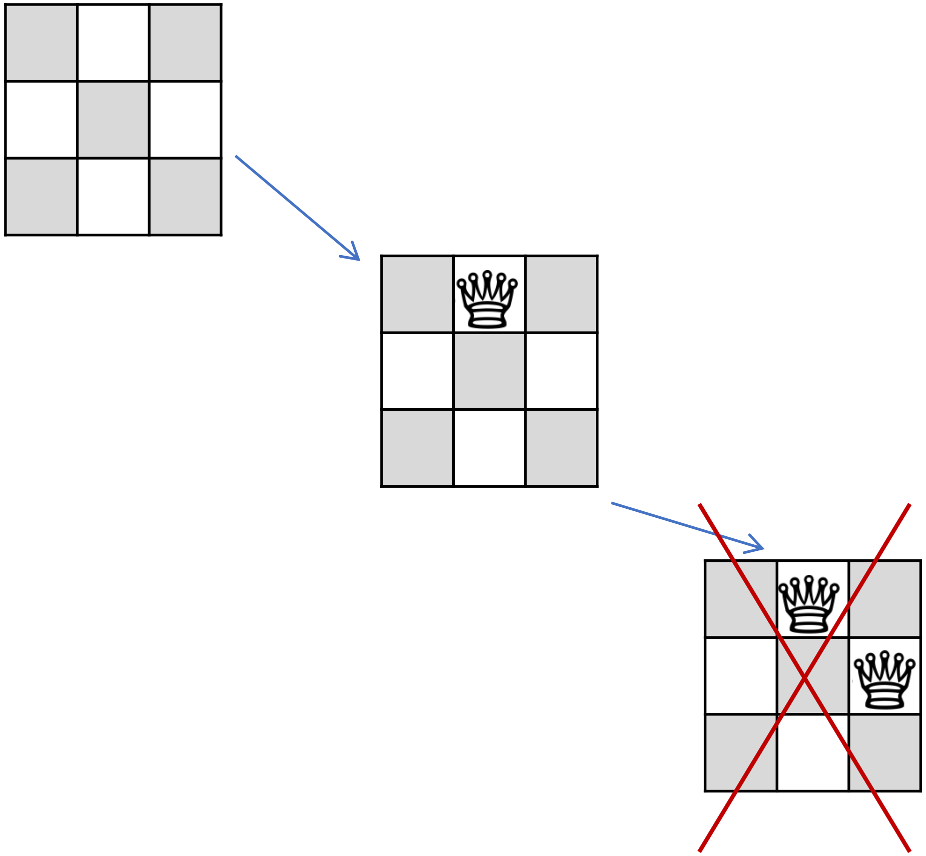


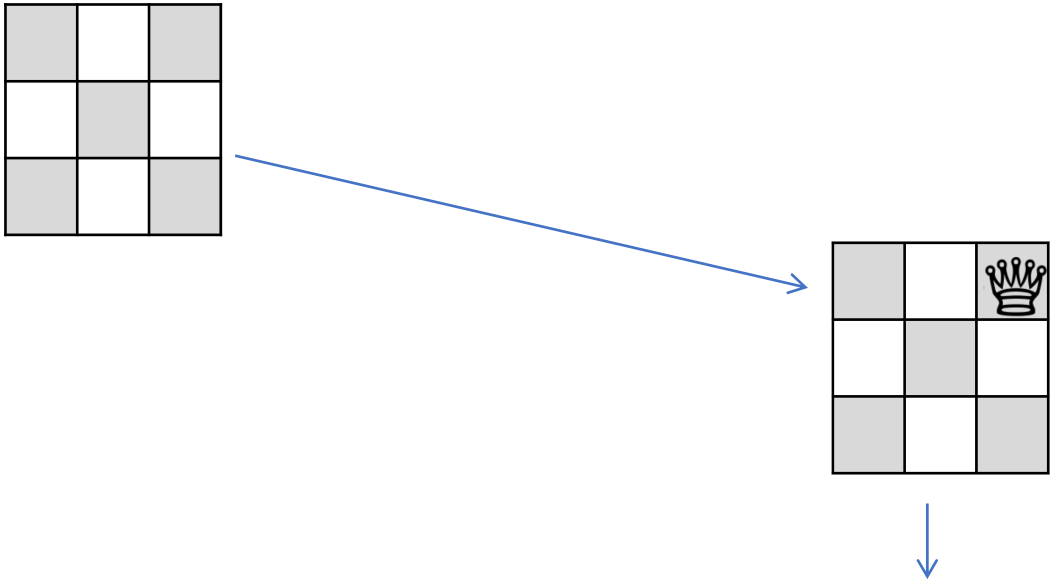












...

Temă

Problema celor n dame

- **Reprezentarea soluţiei**

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Problema celor n dame

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k$ = coloana pe care este plasată dama de pe linia $\mathbf{k}$

  $\mathbf{x}_k \in \{1, 2, \dots, n\}$ $(\mathbf{p}_k = 1,\ \mathbf{u}_k = \mathbf{n})$

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Problema celor n dame

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k$ = coloana pe care este plasată dama de pe linia $\mathbf{k}$

  $\mathbf{x}_k \in \{1, 2, \ldots, n\}$ $(\mathbf{p}_k = 1,\ \mathbf{u}_k = n)$

- **Condiţii interne (finale)**

  pentru orice $\mathbf{i} \neq \mathbf{j}$: $\mathbf{x}_i \neq \mathbf{x}_j$ și $|\mathbf{x}_i - \mathbf{x}_j| \neq |\mathbf{j} - \mathbf{i}|$

- **Condiţii de continuare**

Problema celor n dame

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k$ = coloana pe care este plasată dama de pe linia $\mathbf{k}$

  $\mathbf{x}_k \in \{1, 2, \ldots, n\}$ $(\mathbf{p}_k = 1,\ \mathbf{u}_k = n)$

- **Condiţii interne (finale)**

  pentru orice $\mathbf{i} \neq \mathbf{j}$: $\mathbf{x}_i \neq \mathbf{x}_j$ și $|\mathbf{x}_i - \mathbf{x}_j| \neq |\mathbf{j} - \mathbf{i}|$

- **Condiţii de continuare** – pentru x_k

  pentru orice $\mathbf{i} < \mathbf{k}$: $\mathbf{x}_i \neq \mathbf{x}_k$ și $|\mathbf{x}_i - \mathbf{x}_k| \neq \mathbf{k} - \mathbf{i}$

Implementare - varianta recursivă

```
boolean cont(int k){
   for(int i=1;i<k;i++)
     if((x[i]==x[k]) || (Math.abs(x[k]-x[i])==k-i))
          return false;
     return true;
}
```

Implementare - varianta recursivă

```
boolean cont(int k){
   for(int i=1;i<k;i++)
     if((x[i]==x[k]) || (Math.abs(x[k]-x[i])==k-i))
          return false;
     return true;
}
void retsol(int[] x){
   for(int i=1;i<=n;i++)
      System.out.print("("+i+","+x[i]+") ");
   System.out.println();
}
```

Implementare - varianta recursivă

```
void backrec(int k){
    if(k==n+1)
        retsol(x);
    else
        for(int i=1;i<=n ;i++){ //Xk
          x[k]=i;
          if (cont(k))
            backrec(k+1);
        }
}
```

Implementare - varianta nerecursivă

```
void back(){
  int k=1;
  x=new int[n+1];
  for(int i=1;i<=n;i++) x[i]=0;
  while(k>0){
    if(k==n+1){ retsol(x); k--; }//revenire dupa sol
    else{
        if(x[k]<n){



        }
        else{                    }   //revenire
    }
  }
}
```

Implementare - varianta nerecursivă

```
void back(){
   int k=1;
   x=new int[n+1];
   for(int i=1;i<=n;i++) x[i]=0;
   while(k>0){
     if(k==n+1){ retsol(x); k--; }//revenire dupa sol
     else{
         if(x[k]<n){
           x[k]++;                   //atribuie
           if (cont(k)) k++;         //si avanseaza
         }
         else{ x[k]=0;  k--; }    //revenire
     }
   }
}
```

Șiruri corecte de paranteze

- Să se genereze toate şirurile de n paranteze ce se închid corect (n par)

Șiruri corecte de paranteze

- Să se generereze toate şirurile de n paranteze ce se închid corect (n par)

- Numărul de şiruri corecte = $C_{n/2}$
  (numerele lui Catalan – v. PD)

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k \in \{\texttt{'('}, \texttt{')'}\}$

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k \in \{'(', ')'\}$

- **Condiţii interne (finale)**

  Notăm $\mathbf{dif} = \mathbf{nr}_(- \mathbf{nr}_)$

  $\mathbf{dif} = 0$

  $\mathbf{dif} \geq 0$ pentru orice secvenţă $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$

- **Condiţii de continuare**

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k \in \{\texttt{'('}, \texttt{')'}\}$

- **Condiţii interne (finale)**

  Notăm $\mathbf{dif} = \mathbf{nr}_(- \mathbf{nr}_)$

  $\mathbf{dif} = 0$

  $\mathbf{dif} \geq 0$ pentru orice secvenţă $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$

- **Condiţii de continuare**

  $\mathbf{dif} \geq 0$ -> doar necesar

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

  $\mathbf{x}_k \in \{'(', ')'\}$

- **Condiţii interne (finale)**

  Notăm $\mathbf{dif} = \mathbf{nr}_(- \mathbf{nr}_)$

  $\mathbf{dif} = 0$

  $\mathbf{dif} \geq 0$ pentru orice secvenţă $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$

- **Condiţii de continuare**

  $\mathbf{dif} \geq 0$ -> doar necesar

  $\mathbf{dif} \leq \mathbf{n-k}$ -> şi suficient

```
void back(){
   dif=0;
   back(1);
}
void back(int k){
   if(k==n+1)
        retsol(x);
   else{









   }
}
```

```
void back(){
   dif=0;
   back(1);
}
void back(int k){
   if(k==n+1)
        retsol(x);
   else{
        x[k]='(';
        dif++;
        if (dif <= n-k)
           back(k+1);
        dif--;




   }
}
```

```
void back(){
   dif=0;
   back(1);
}
void back(int k){
   if(k==n+1)
        retsol(x);
   else{
        x[k]='(';
        dif++;
        if (dif <= n-k)
           back(k+1);
        dif--;

        x[k]=')';
        dif--;
        if (dif >= 0)
           back(k+1);
        dif++;
   }
}
```

Șiruri corecte de paranteze

- **Implementare nerecursivă** -temă

Metoda Backtracking

- Fie vectorul $a=(a_1,...,a_n)$. Să se determine toate subşirurile crescătoare de lungime maximă.

Metoda Backtracking

- Fie vectorul a=(a_1,...,a_n). Să se determine toate subşirurile crescătoare de lungime maximă.

- **Subproblemă:**

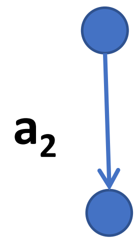
  `lung[i]` = lungimea maximă a unui subşir crescător ce începe pe poziția i

- **Soluție problemă:**

  $$\mathtt{lmax} = \max\{\mathtt{lung[i]} \mid \mathtt{i} = 1,2,\ldots,\mathtt{n}\}$$

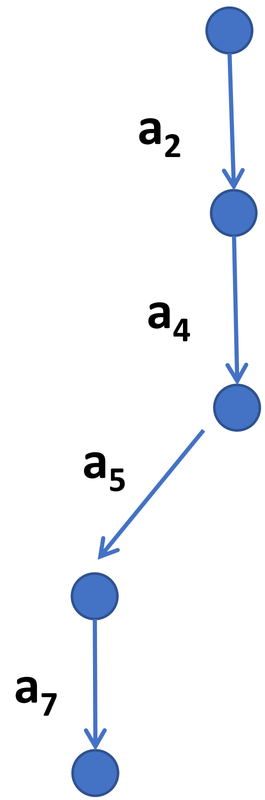
Subşirurile crescătoare maxime

a:	**8**	**1**	**7**	**4**	**6**	**5**	**11**
	1	2	3	4	5	6	7
lung :	**2**	**4**	**2**	**3**	**2**	**2**	**1**



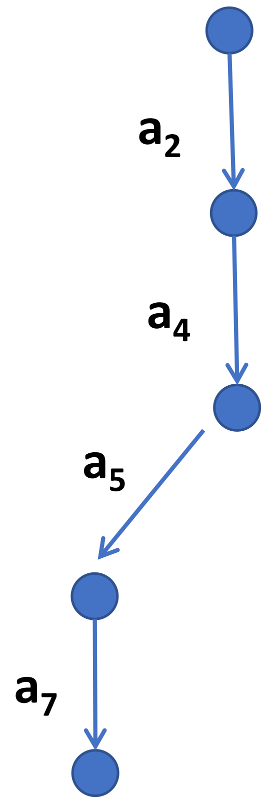
Subşirurile crescătoare maxime

a:	**8**	**1**	**7**	**4**	**6**	**5**	**11**
	1	2	3	4	5	6	7
lung :	**2**	**4**	**2**	**3**	**2**	**2**	**1**



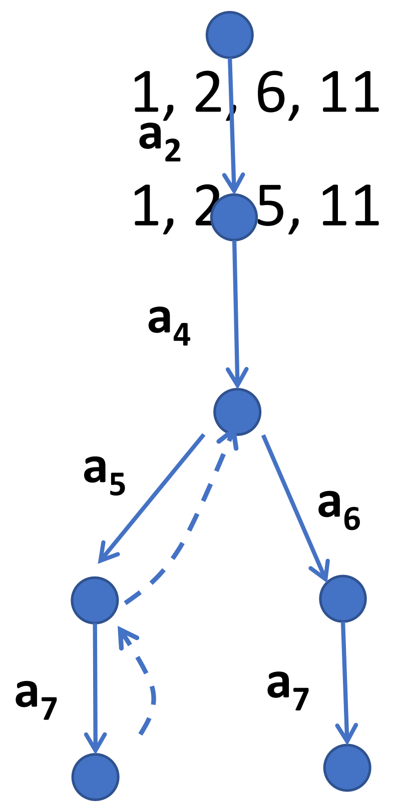
Subşirurile crescătoare maxime

a:	**8**	**1**	**7**	**4**	**6**	**5**	**11**
	1	2	3	4	5	6	7
lung :	**2**	**4**	**2**	**3**	**2**	**2**	**1**



Metoda Backtracking

a:	**8**	**1**	**7**	**4**	**6**	**5**	**11**
	1	2	3	4	5	6	7
lung :	**2**	**4**	**2**	**3**	**2**	**2**	**1**



Metoda Backtracking

- **Reprezentarea soluţiei**

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Metoda Backtracking

- **Reprezentarea soluţiei**

  $x = \{x_1, x_2, \ldots, x_{lmax}\}$, unde

  $x_k \in \{1, \ldots, n\}$ poziţie din vectorul a

- **Condiţii interne (finale)**

  $ax_1 < ax_2 < \ldots < ax_{lmax}$

- **Condiţii de continuare**

Metoda Backtracking

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{x_1, x_2, \dots, x_{lmax}\}$, unde

  $x_k \in \{1, \dots, n\}$ poziţie din vectorul a

- **Condiţii interne (finale)**

  $ax_1 < ax_2 < \dots < ax_{lmax}$

- **Condiţii de continuare**

  `lung[a[x`$_1$`]] = lmax`

  $x_{k-1} < x_k$, $ax_{k-1} < ax_k$, `lung[x`$_k$`]=lung[x`$_{k-1}$`]-1`

```
void scrie(int k){
     if (k==lmax+1){    //retsol();
          for (int i=1;i<=lmax;i++)
                       System.out.print(a[x[i]]+" ");
            System.out.println();
     }
     else
          for (int j=x[k-1]+1;j<=n;j++){
               x[k]=j;




            }
}
```

```
for(int i=1;i<=n;i++)
     if (lung[i]==lmax){
            x[1]=i;
            scrie(2);
     }
```

```
void scrie(int k){
    if (k==lmax+1){   //retsol();
        for (int i=1;i<=lmax;i++)
                System.out.print(a[x[i]]+" ");
          System.out.println();
    }
    else
        for (int j=x[k-1]+1;j<=n;j++){
            x[k]=j;
            if ((a[x[k-1]]<a[x[k]]) &&
                    (lung[x[k-1]]==1+lung[x[k]]))
                    scrie(k+1);
          }
}
```

```
for(int i=1;i<=n;i++)
    if (lung[i]==lmax){
          x[1]=i;
          scrie(2);
    }
```

Metoda Backtracking - SAT

- **Problema satisfiabilităţii SAT**

Considerăm o expresie logică în care apar variabilele $\{x_1, x_2, \ldots, x_n\}$ şi negaţiile lor $\overline{x_i}$. Ştiind că expresia este de forma

$$E = C_1 \land C_2 \land \ldots \land C_m$$

unde C_i sunt clauze disjunctive = în care apare doar operatorul $\lor$, să se verifice dacă se pot atribui valori variabilelor astfel încât valoarea expresiei să fie true (expresia să fie satisfăcută)

Metoda Backtracking - SAT

- **Problema satisfiabilităţii SAT**

Considerăm o expresie logică în care apar variabilele $\{x_1, x_2, \ldots, x_n\}$ şi negaţiile lor $\overline{x_i}$. Ştiind că expresia este de forma

$$E = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

unde C_i sunt clauze disjunctive = în care apare doar operatorul $\vee$, să se verifice dacă se pot atribui valori variabilelor astfel încât valoarea expresiei să fie true (expresia să fie satisfăcută)

Literal = x_i sau $\overline{x_i}$

FNC – formă normală conjunctivă

Metoda Backtracking - SAT

- **Problema satisfiabilităţii SAT**

Considerăm o expresie logică în care apar variabilele $\{x_1, x_2, \dots, x_n\}$ şi negaţiile lor $\overline{x_i}$. Ştiind că expresia este de forma

$$E = C_1 \land C_2 \land \dots \land C_m$$

unde C_i sunt clauze disjunctive = în care apare doar operatorul $\lor$, să se verifice dacă se pot atribui valori variabilelor astfel încât valoarea expresiei să fie true (expresia să fie satisfăcută)

- **k-SAT – fiecare clauză are cel mult k literali**

Metoda Backtracking - SAT

- **Problema satisfiabilităţii SAT**

$$E = (x_1 \lor \overline{x_2}) \land (x_1 \lor x_2 \lor x_3 \lor x_4) \land (\overline{x_1} \lor x_3) \land (\overline{x_4} \lor \overline{x_3})$$

adevărată pentru $x_1 = x_2 = x_3 = 1$, $x_4 = 0$

- k-SAT = fiecare clauză are cel mult k literali
- k = 2 - polinomial
- k = 3 – NP-completă

Metoda Backtracking - SAT

- **Generăm toate şirurile binare $x_1x_2…x_n$ (0 = false, 1 = true) şi verificăm pentru fiecare astfel de şir înlocuind în expresia E dacă E devine adevărată**
- Dacă găsim un şir pentru care expresia este adevărată oprim generarea

Metoda Backtracking - SAT

- **Generăm toate şirurile binare $x_1x_2 \ldots x_n$ (0 = false, 1 = true) şi verificăm pentru fiecare astfel de şir înlocuind în expresia E dacă E devine adevărată**
- Dacă găsim un şir pentru care expresia este adevărată oprim generarea
    - **Complexitate?**

Metoda Backtracking - SAT

- **Generăm toate şirurile binare $x_1x_2 \ldots x_n$ (0 = false, 1 = true) şi verificăm pentru fiecare astfel de şir înlocuind în expresia E dacă E devine adevărată**
- Dacă găsim un şir pentru care expresia este adevărată oprim generarea

  - Dacă expresia nu poate fi satisfăcută, atunci se vor genera şi testa toate şirurile binare de lungime n

  $O(2^n m)$

Metoda Backtracking - SAT

- **Putem face verificări pe parcurs ⇒ condiţii de continuare?**
- **Contează pentru performanţă ordinea în care dăm valori variabilelor? ⇒ euristici**

Metoda Backtracking - SAT

- **Condiţii de continuare**

Înlocuim valoarea v dată variabilei x_k la pasul k în fiecare clauză C_i din E în care apare x_k sau $\overline{x_k}$.

În clauză literalul corespunzător lui x_k este în una din situaţiile:

- devine 1 ⇒

- devine 0 ⇒

Metoda Backtracking - SAT

- **Condiţii de continuare**

Înlocuim valoarea v dată variabilei x_k la pasul k în fiecare clauză C_i din E în care apare x_k sau $\overline{x_k}$.

În clauză literalul corespunzător lui x_k este în una din situaţiile:

- devine 1 ⇒ pot elimina clauza C_i din E

- devine 0 ⇒ pot elimina 0 (literalul corespunzător lui x_k) din C_i

Metoda Backtracking - SAT

- **Condiţii de continuare**

Înlocuim valoarea v dată variabilei x_k la pasul k în fiecare clauză C_i din E în care apare x_k sau $\overline{x_k}$.

În clauză literalul corespunzător lui x_k este în una din situaţiile:

- devine 1 $\Rightarrow$ pot elimina clauza C_i din E

- devine 0 $\Rightarrow$ pot elimina 0 (literalul corespunzător lui x_k) din C_i

Când nu mai are sens să continuăm (pentru că expresia nu poate fi satisfăcută cu valorile date deja variabilelor)?

Metoda Backtracking - SAT

- **Condiţii de continuare**

Înlocuim valoarea v dată variabilei x_k la pasul k în fiecare clauză C_i din E în care apare x_k sau $\overline{x_k}$.

În clauză literalul corespunzător lui x_k este în una din situaţiile:

- devine 1 ⇒ pot elimina clauza C_i din E

- devine 0 ⇒ pot elimina 0 (literalul corespunzător lui x_k) din C_i

Dacă o clauză devine vidă, dar expresia nu este vidă, atunci încercarea de a da valoarea v lui x_k este eşuată

Metoda Backtracking - SAT

- **Condiţii de continuare**

Înlocuim valoarea v dată variabilei x_k la pasul k în fiecare clauză C_i din E în care apare x_k sau $\overline{x_k}$.

În clauză literalul corespunzător lui x_k este în una din situaţiile:

- devine 1 ⇒ pot elimina clauza C_i din E

- devine 0 ⇒ pot elimina 0 (literalul corespunzător lui x_k) din C_i

Dacă expresia E devine vidă atunci am găsit o soluţie (restul variabilelor pot primi orice valoare)

Metoda Backtracking - SAT

- **Temă – backtracking pentru 3-SAT**

(n variabile, m clauze de lungime cel mult 3 = cu cel mult 3 literali)

Metoda Backtracking - SAT

$(x_2 \lor \overline{x_3}) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2})$

x_1:

0

Metoda Backtracking - SAT

$(x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2})$

x_1: 0

$(x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3) \wedge (\overline{x_2})$

x_2: 0

Metoda Backtracking - SAT

$(x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2})$

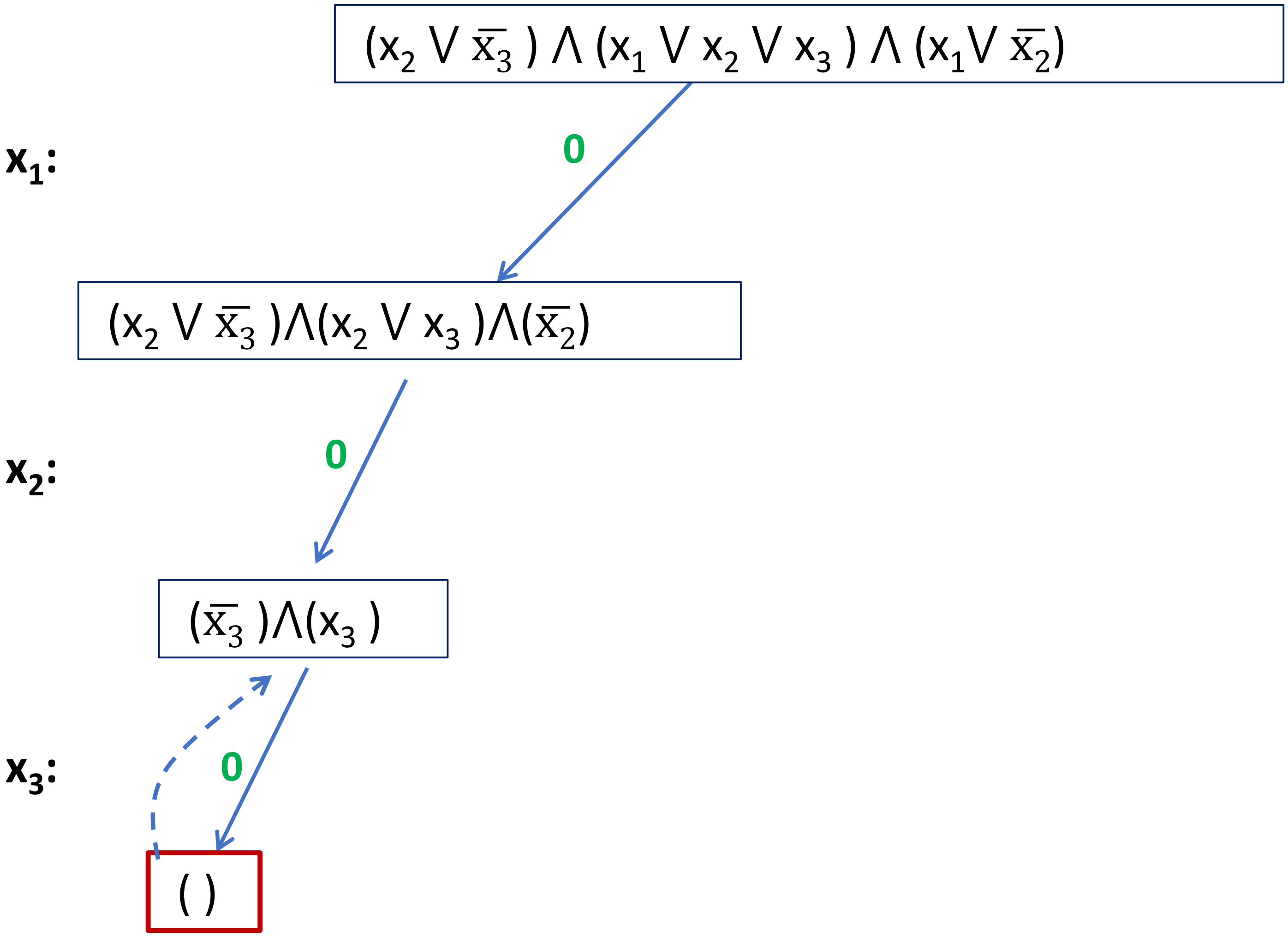
x_1: $\quad 0$

$(x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3) \wedge (\overline{x_2})$

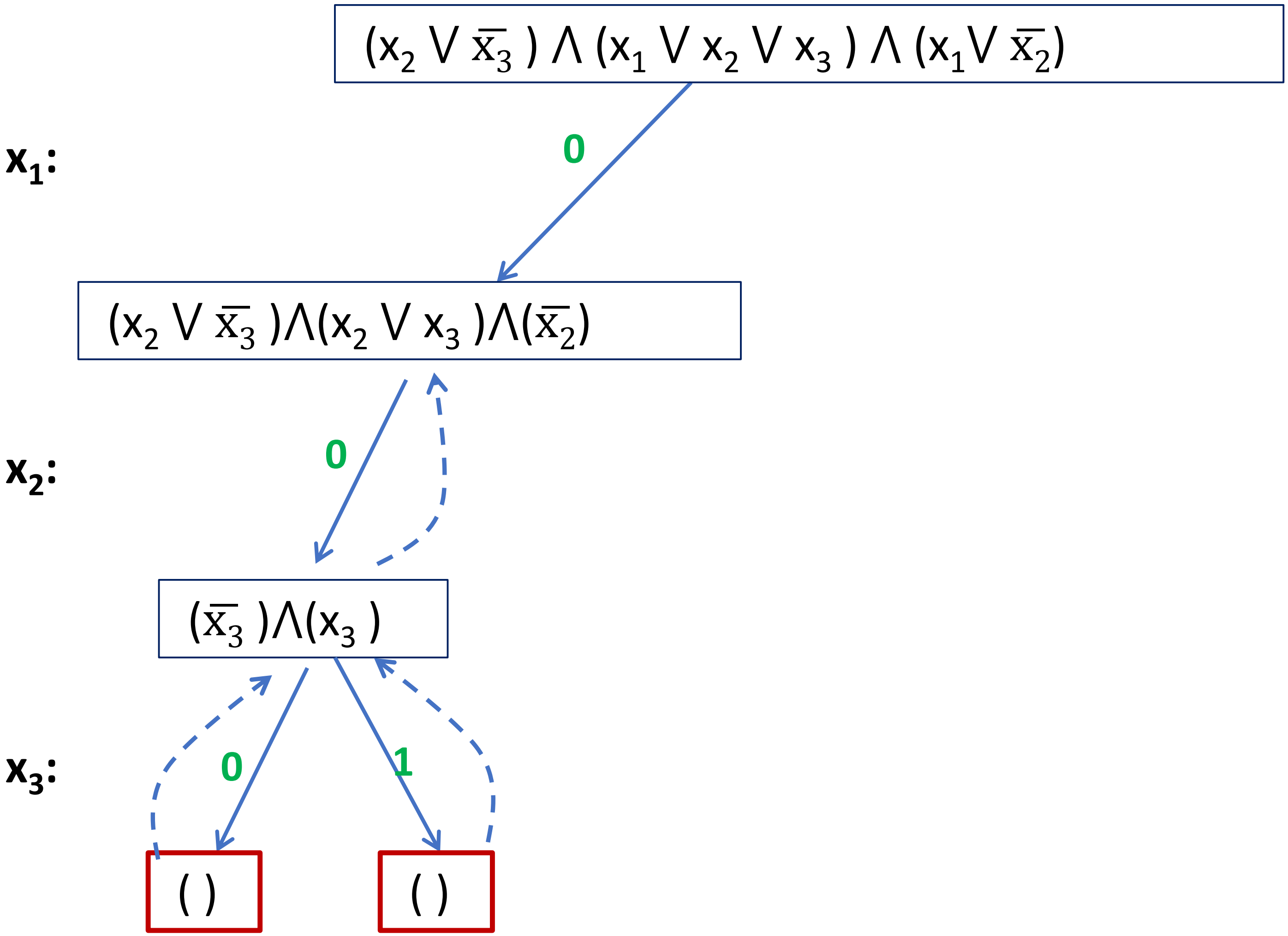
x_2: $\quad 0$

$(\overline{x_3}) \wedge (x_3)$

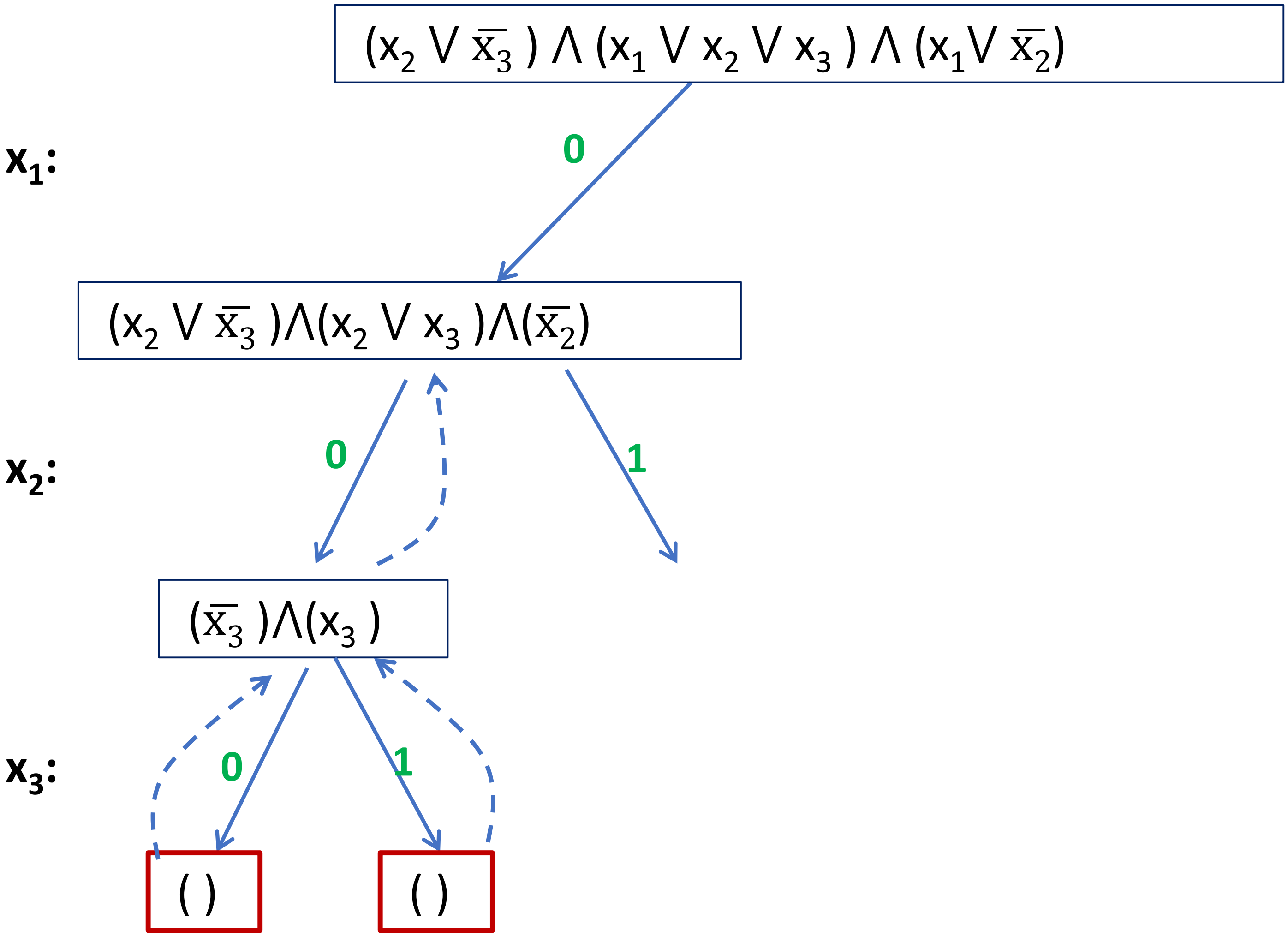
Metoda Backtracking - SAT



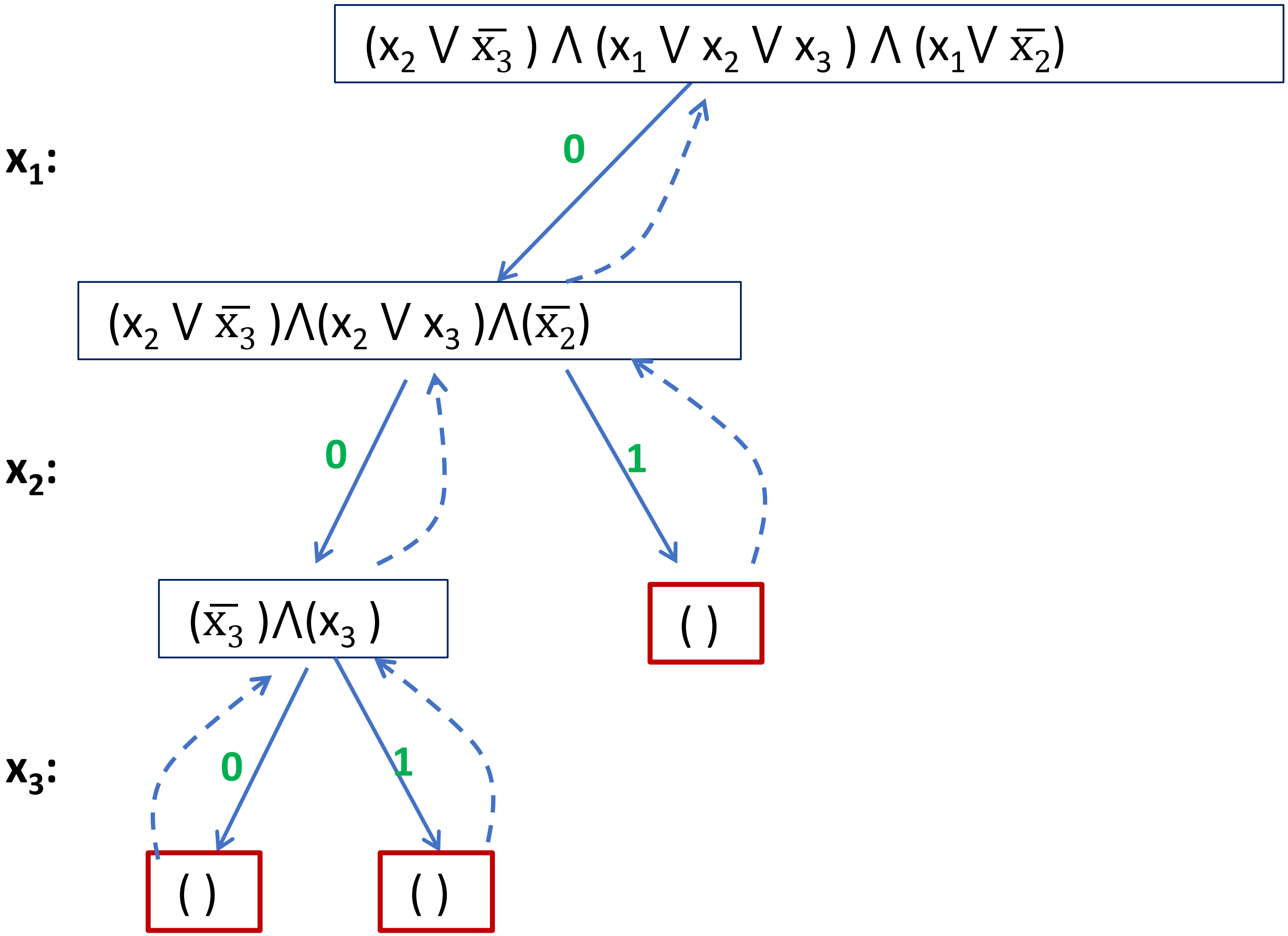
Metoda Backtracking - SAT



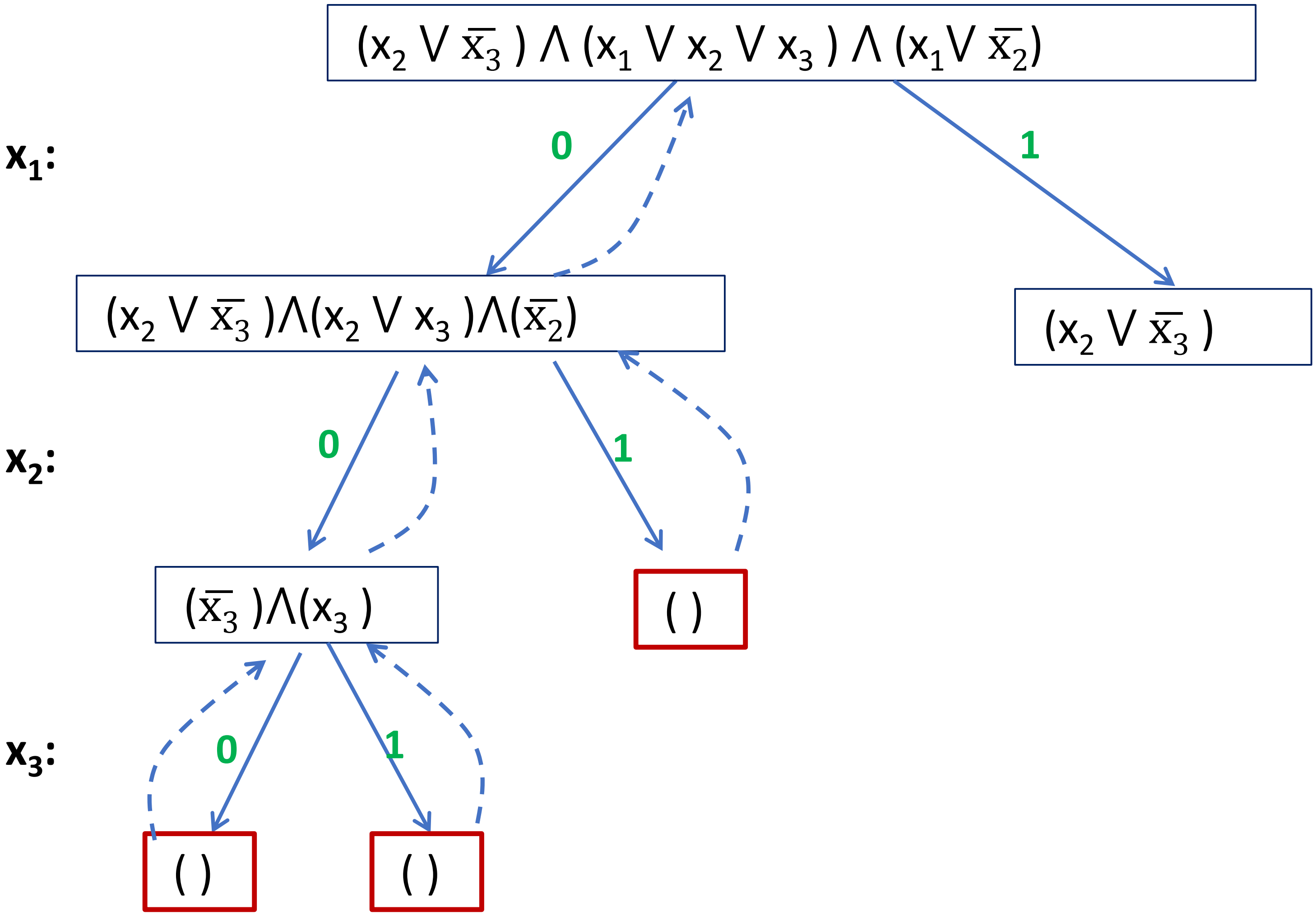
Metoda Backtracking - SAT



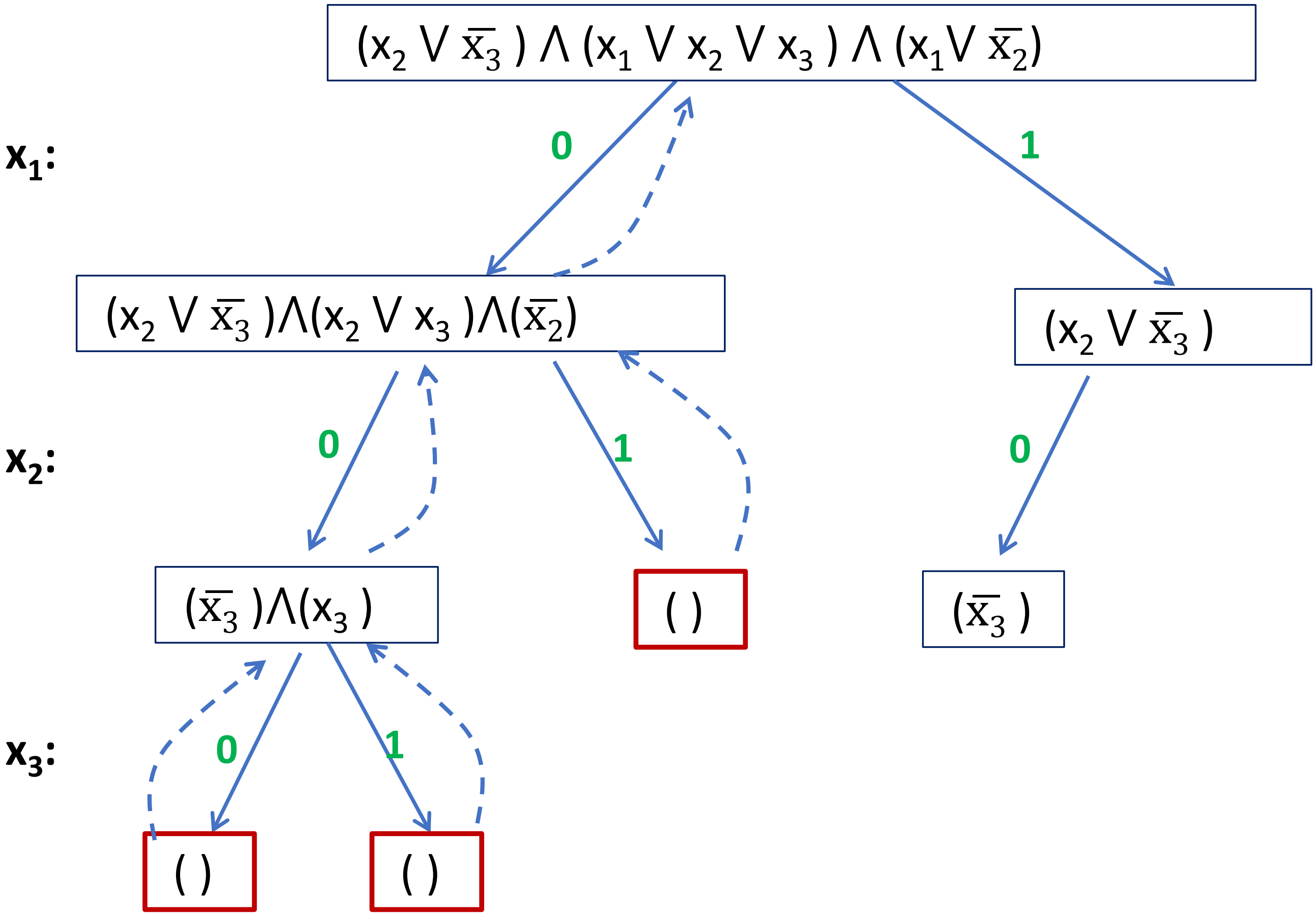
Metoda Backtracking - SAT



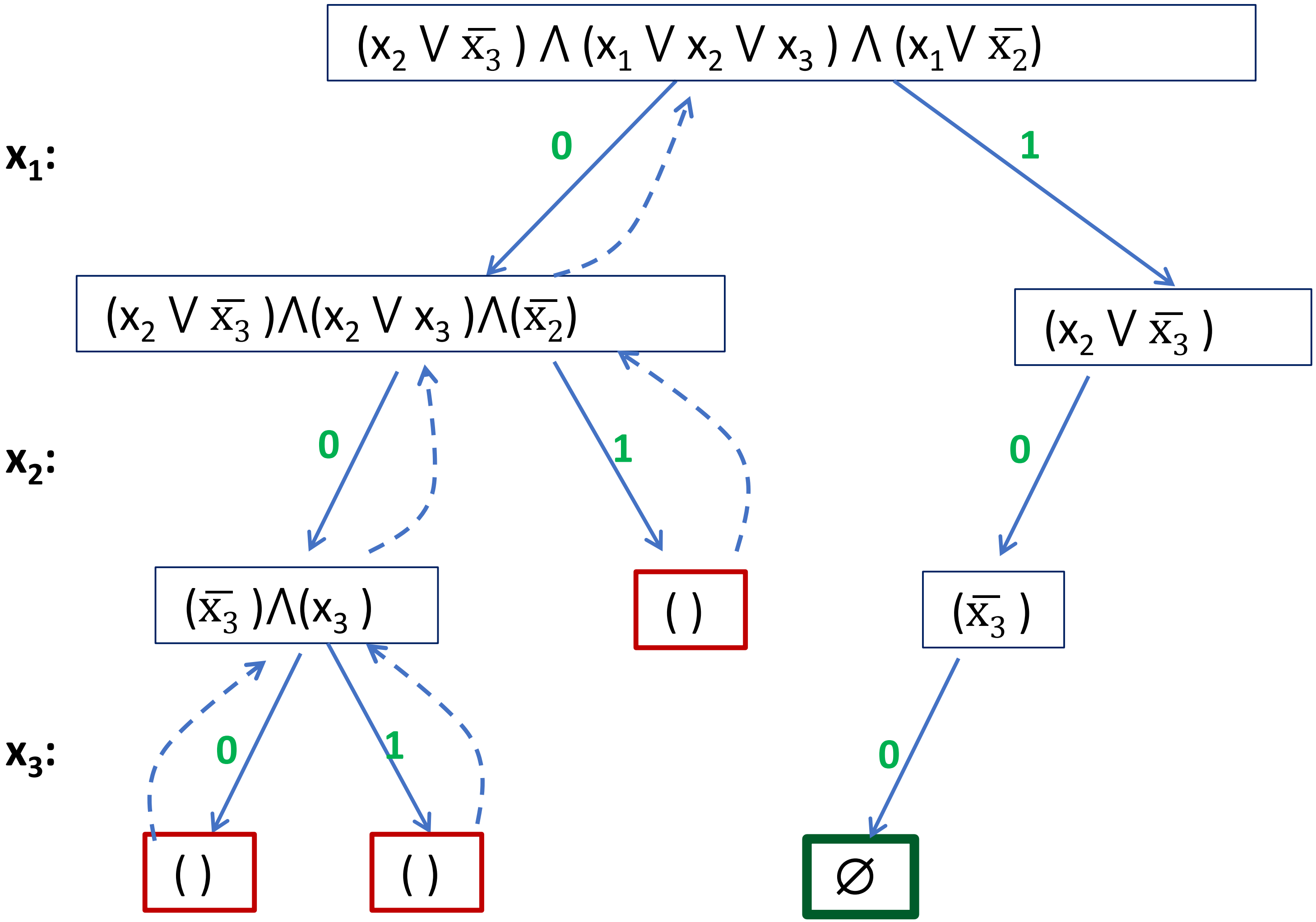
Metoda Backtracking - SAT



Metoda Backtracking - SAT



Metoda Backtracking - SAT



E= $\varnothing$ $\Rightarrow$Soluţie + STOP

```
procedure back(k, E, V)
   if E = ∅
         scrie true, x STOP
     x_k ← false;
   for (C clauza în E care contine x_k)
       daca x̄_k ∈ C atunci
            elimina(E, C)
            back(k+1,E)
            adauga(E, C)
```

```
back(1)

scrie false
```

```
procedure back(k, E, V)
   if E = ∅
         scrie true, x STOP
     xk←false;
   for (C clauza în E care contine xk)
       daca x̄k ∈ C atunci
            elimina(E, C)
            back(k+1,E)
            adauga(E, C)
       altfel
            reducem(C, k) – eliminam xk din C
            if C ≠ ∅
               back(k+1,E)
             restauram(C, k) – reintroducem xk in C
```

back(1)

scrie false

```
procedure back(k, E, V)
   if E = ∅
         scrie true, x STOP
     xk←false;
   for (C clauza în E care contine xk)
       daca x̄k ∈ C atunci
            elimina(E, C)
            back(k+1,E)
            adauga(E, C)
       altfel
            reducem(C, k) – eliminam xk din C
            if C ≠ ∅
               back(k+1,E)
             restauram(C, k) – reintroducem xk in C
  xk←true;
```

... SIMILAR

```
back(1)

scrie false
```

Metoda Backtracking - SAT

- **Ordinea în care se dau valori variabilelor**

⇒ euristici

Exemplu

– întâi dam valori **variabilelor care apar în clauze scurte**

- Greedy: literalul care satisface mai multe clauze

- **Detectarea de conflicte + formule de logică**

```
procedure back(k, E, V)  V-multimea variabilelor neselectate
   if E = ∅
         scrie true, x STOP
   t ← alegeVariabila(V)
     x_t←false;
   for (C clauza în E care contine x_t)
       daca x̄_t ∈ C atunci
            elimina(E, C)
            back(k+1,E)
            adauga(E, C)
       altfel
            reducem(C, k) – eliminam x_t din C
            if C ≠ ∅
               back(k+1,E)
             restauram(C, k) – reintroducem x_t in C
  x_t←true;
 ... SIMILAR
```

```
back(1, E, {1,2,…,n})

scrie false
```

Metoda Backtracking

- **Variantele** cele mai uzuale întâlnite în aplicarea metodei backtracking sunt următoarele:
  - soluţia poate avea un număr variabil de componente *şi/sau*
  - dintre ele alegem una care optimizează o funcţie dată

Metoda Backtracking

- **Exemplu**: Dat un număr natural n, să se genereze toate partițiile lui n ca sumă de numere pozitive

Partiție a lui n = $\{x_1, x_2, \ldots, x_k\}$ cu

$$x_1 + x_2 + \ldots + x_k = n$$

```
4 = 1+1+1+1

4 = 1+1+2

4 = 1+3

4 = 2+2
```

Metoda Backtracking - Partiţii

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$, unde

  $\mathbf{x}_i \in \{1, \ldots, \mathbf{n}\}$

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Metoda Backtracking - Partiţii

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$, unde

  $\mathbf{x}_i \in \{1, \ldots, n\}$

- **Condiţii interne (finale)**

  $\mathbf{x}_1 + \mathbf{x}_2 + \ldots + \mathbf{x}_k = \mathbf{n}$

  Pentru unicitate: $\mathbf{x}_1 \leq \mathbf{x}_2 \leq \ldots \leq \mathbf{x}_k$

- **Condiţii de continuare**

Metoda Backtracking - Partiţii

- **Reprezentarea soluţiei**

  $x = \{x_1, x_2, \ldots, x_k\}$, unde

  $x_i \in \{1, \ldots, n\}$

- **Condiţii interne (finale)**

  $x_1 + x_2 + \ldots + x_k = n$

  Pentru unicitate: $x_1 \le x_2 \le \ldots \le x_k$

- **Condiţii de continuare**

  $x_{k-1} \le x_k \longrightarrow x_k \in \{x_{k-1}, \ldots, n\} = X_k$

  $x_1 + x_2 + \ldots + x_k \le n$

Implementare - varianta recursivă

```
void backrec(int k){
    for(int i=x[k-1];i<=n;i++){
        x[k]=i;
        if(s+x[k]<=n)////verif.cond.de cont










        // else return;
    }
}
```

```
void backrec(int k){
    for(int i=x[k-1];i<=n;i++){
        x[k]=i;
        if(s+x[k]<=n)////verif.cond.de cont
            if(s+x[k]==n){//este solutie
                retsol(x,k);
                return;
            }




        // else return;
    }
}
```

```
void backrec(int k){
    for(int i=x[k-1];i<=n;i++){
        x[k]=i;
        if(s+x[k]<=n)////verif.cond.de cont
             if(s+x[k]==n){//este solutie
                 retsol(x,k);
                 return;
            }
            else{
                 s+=x[k];
                backrec(k+1);
                s-=x[k];
            }
        // else return;
    }
}
```

```
void retsol(int[] x,int k){
        for(int i=1;i<=k;i++)
            System.out.print(x[i]+" ");
        System.out.println();
}

 void backrec(){
        x=new int[n+1];
        x[0]=1; //prima valoare pentru x[1]
        s=0;
        backrec(1);
  }
```

Implementare - varianta nerecursivă

```
void back(){
   int k=1,s=0; int x[]=new int[n+1];
   x[1]=0;
   while(k>=1){
      if(x[k]<n){
       x[k]++; s++;
       if(s<=n){//cont - verif. conditiilor de cont











      }
   }
  }
```

```
void back(){
   int k=1,s=0; int x[]=new int[n+1];
   x[1]=0;
   while(k>=1){
        if(x[k]<n){
         x[k]++; s++;
         if(s<=n){//cont - verif. conditiilor de cont
              if(s==n){//dc este sol
                  retsol(x,k);
                  s=s-x[k]; k--;//revenire dupa sol
              }




        }
    }
   }
```

```
void back(){
   int k=1,s=0; int x[]=new int[n+1];
   x[1]=0;
   while(k>=1){
       if(x[k]<n){
        x[k]++; s++;
        if(s<=n){//cont - verif. conditiilor de cont
            if(s==n){//dc este sol
                retsol(x,k);
                s=s-x[k]; k--;//revenire dupa sol
            }
            else{ k++; x[k]=x[k-1]-1; s+=x[k]; //avansare
            }
        }


       }
   }
  }
```

```
void back(){
  int k=1,s=0; int x[]=new int[n+1];
  x[1]=0;
  while(k>=1){
     if(x[k]<n){
      x[k]++; s++;
      if(s<=n){//cont - verif. conditiilor de cont
        if(s==n){//dc este sol
          retsol(x,k);
          s=s-x[k]; k--;//revenire dupa sol
        }
        else{ k++; x[k]=x[k-1]-1; s+=x[k]; //avansare
        }
      }
        else{  s=s-x[k]; k--; //revenire
      }
     }
  }
 }
```

Metoda Backtracking - Partiţii

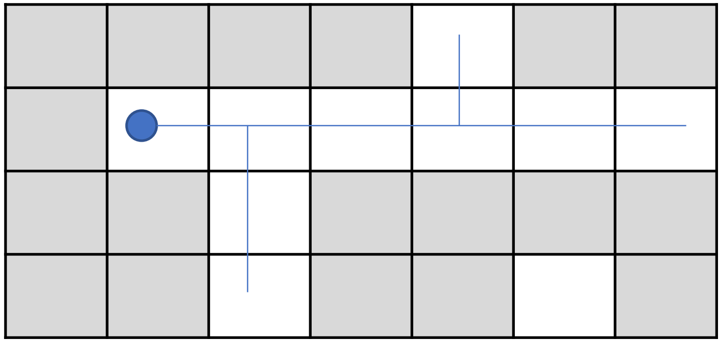
- **DE EVITAT** recalcularea lui s la fiecare pas ca fiind $s = x[1] + \dots + x[k]$

Backtracking în plan

Backtracking în plan

- **Labirint**. Se consideră un caroiaj (matrice) A cu m linii şi n coloane. Poziţiile pot fi:
    - libere: $a_{ij}=0$;
    - ocupate: $a_{ij}=1$.

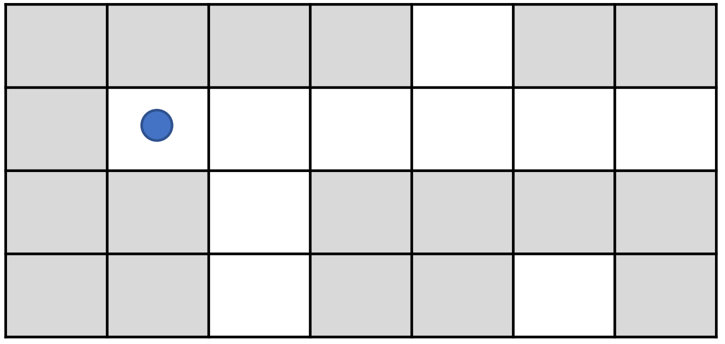
  Se mai dă o poziţie (i_0,j_0). Se caută **toate** drumurile care ies în afara matricei, trecând numai prin poziţii libere (fără a trece de două ori prin aceeaşi poziţie).



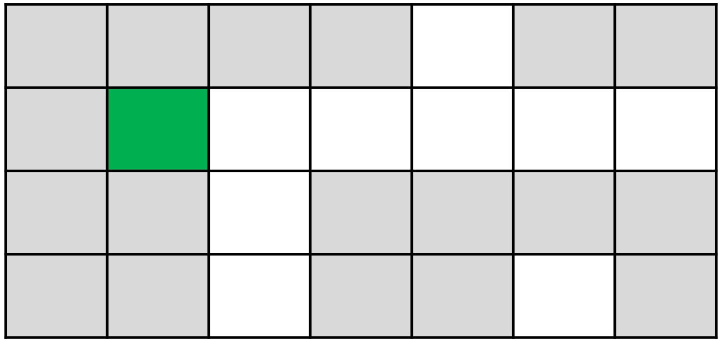
Variante:

- drumul maxim
- drumul minim

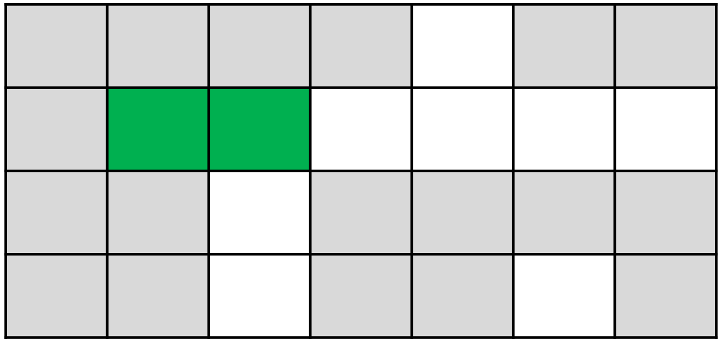
Backtracking în plan



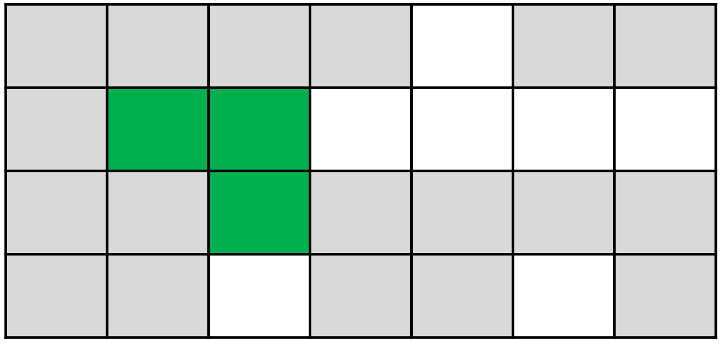
Backtracking în plan



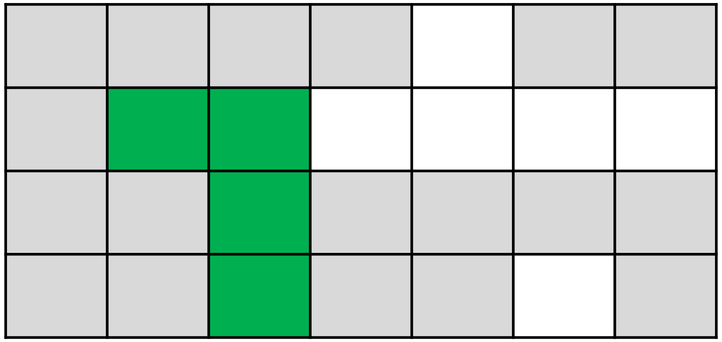
Backtracking în plan



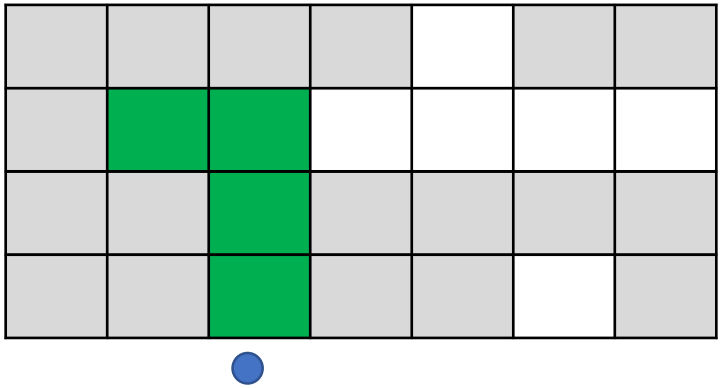
Backtracking în plan



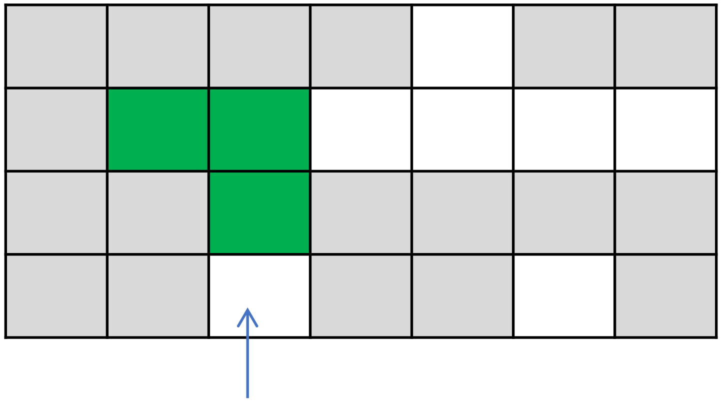
Backtracking în plan



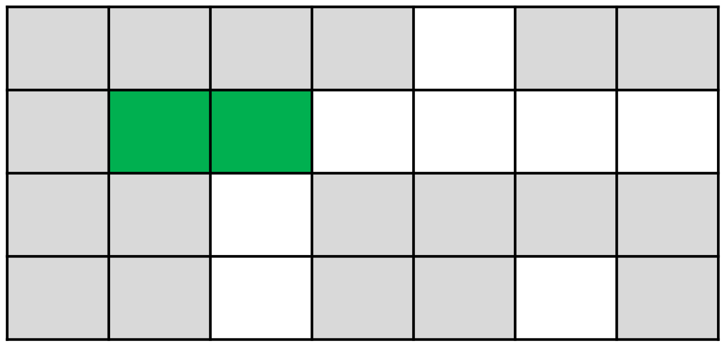
Backtracking în plan



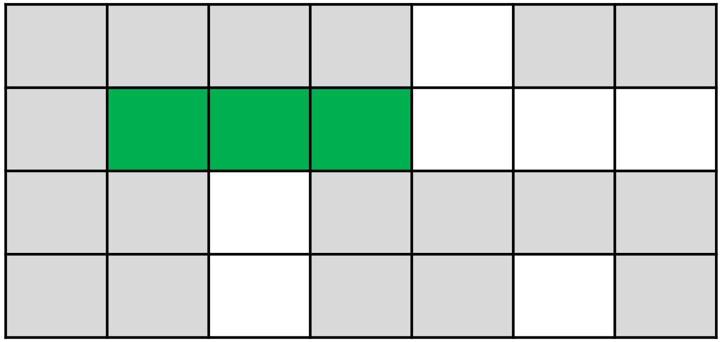
Backtracking în plan



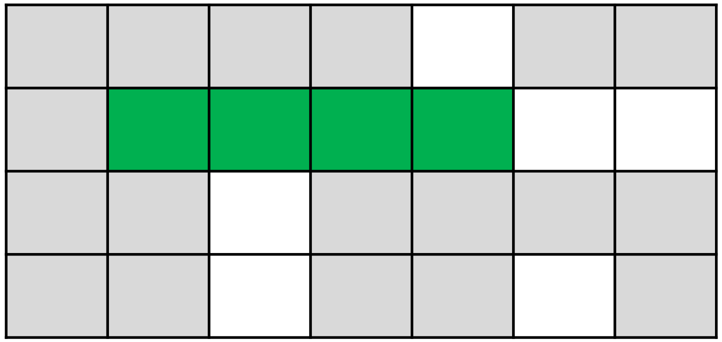
Backtracking în plan



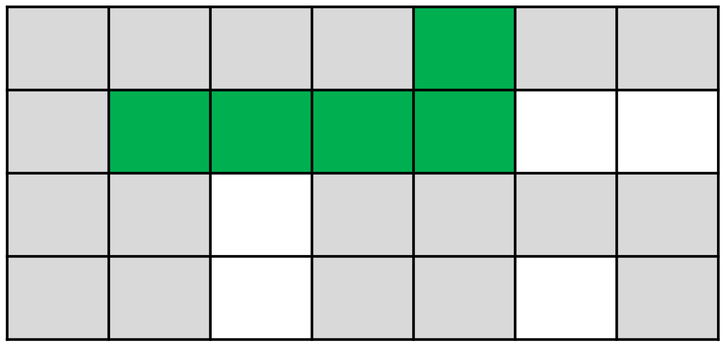
Backtracking în plan



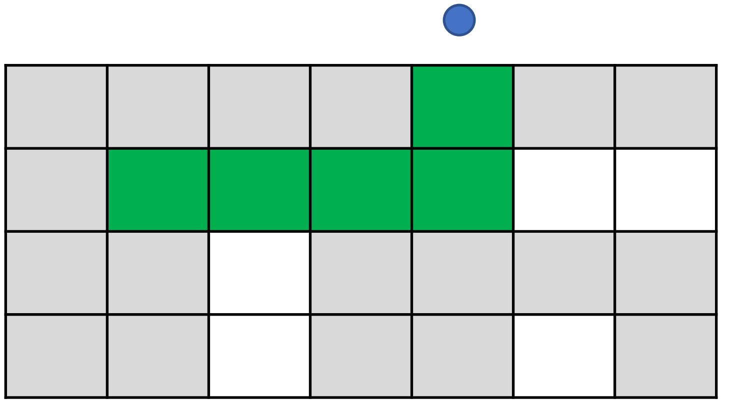
Backtracking în plan



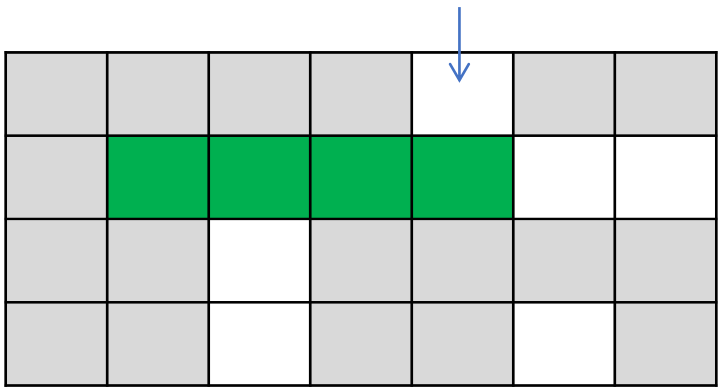
Backtracking în plan



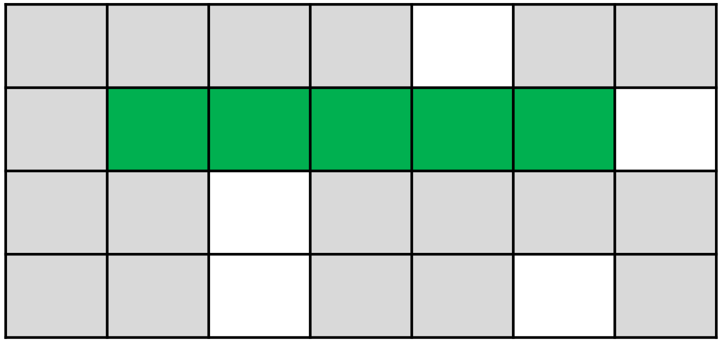
Backtracking în plan



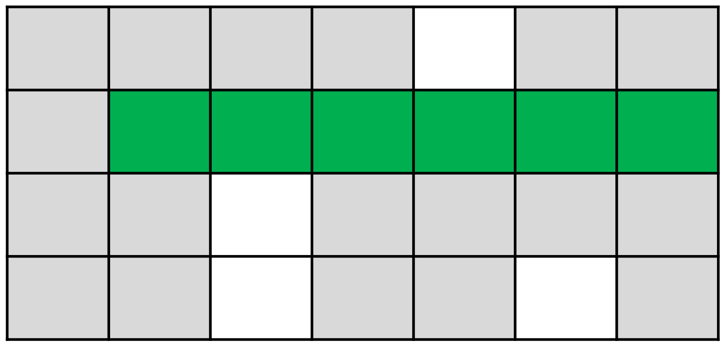
Backtracking în plan



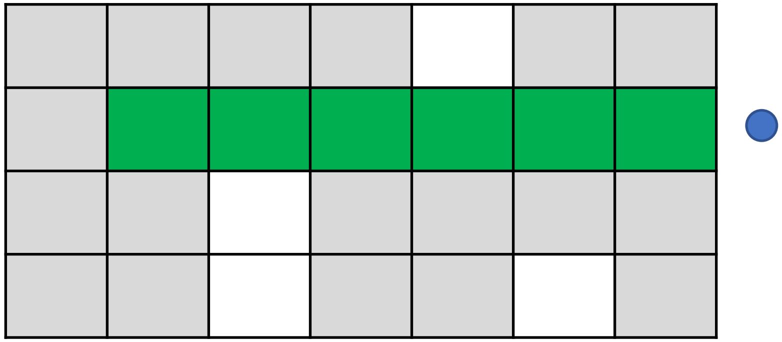
Backtracking în plan



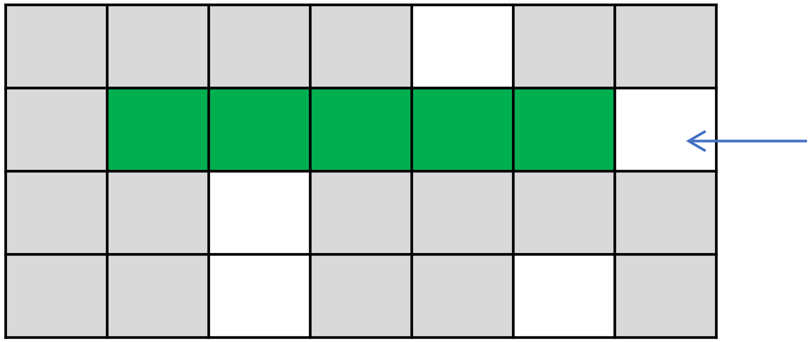
Backtracking în plan



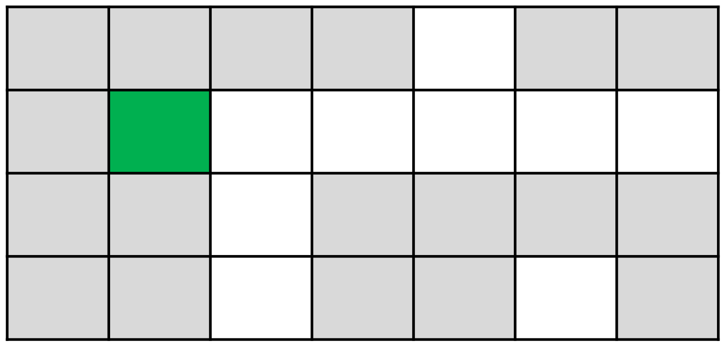
Backtracking în plan



Backtracking în plan



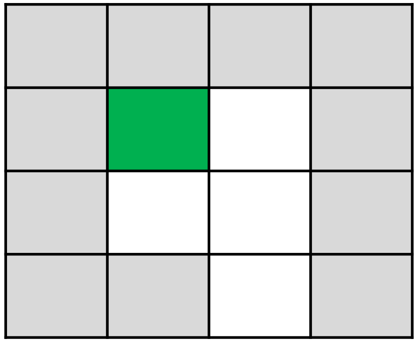
Backtracking în plan



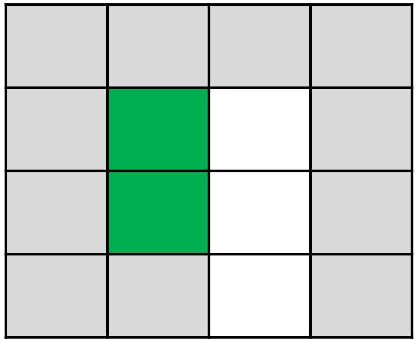
Backtracking în plan

- **Observaţie**: este important să demarcăm celulele când facem pasul înapoi

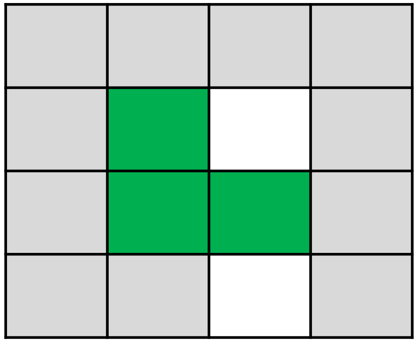
Backtracking în plan



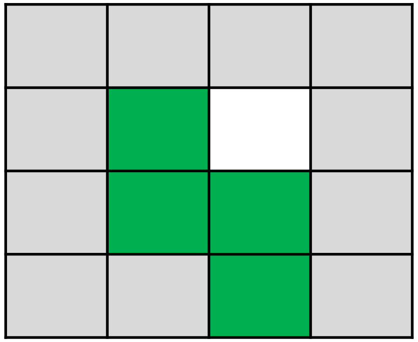
Backtracking în plan



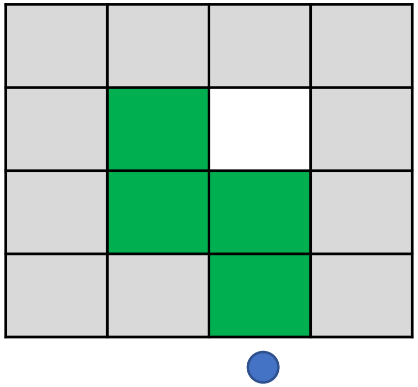
Backtracking în plan



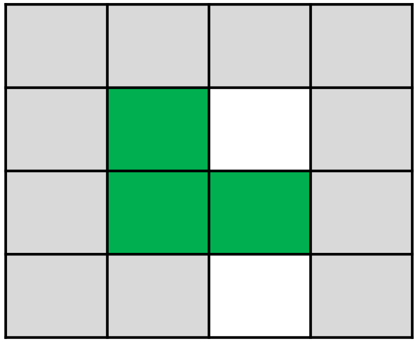
Backtracking în plan



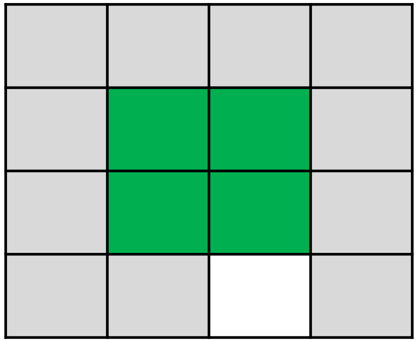
Backtracking în plan



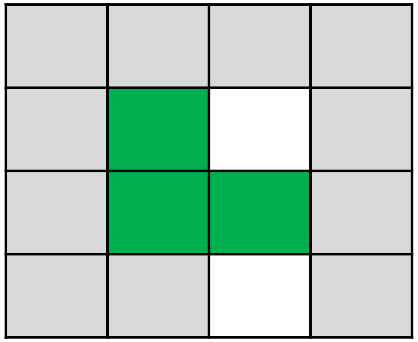
Backtracking în plan



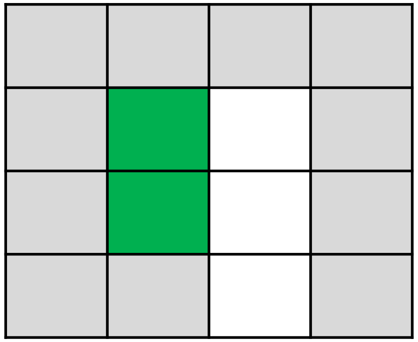
Backtracking în plan



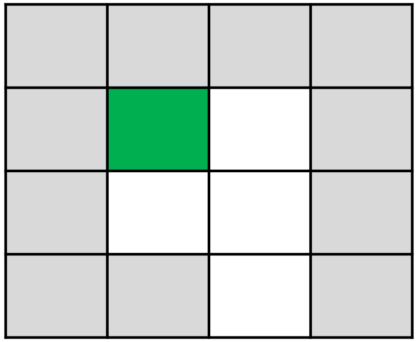
Backtracking în plan



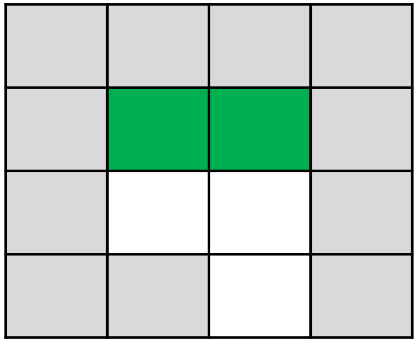
Backtracking în plan



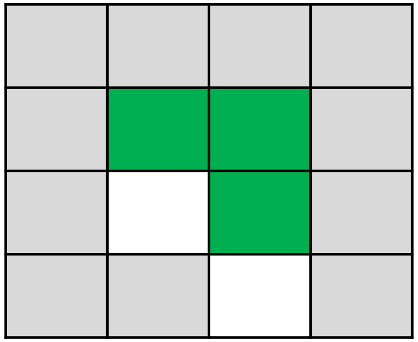
Backtracking în plan



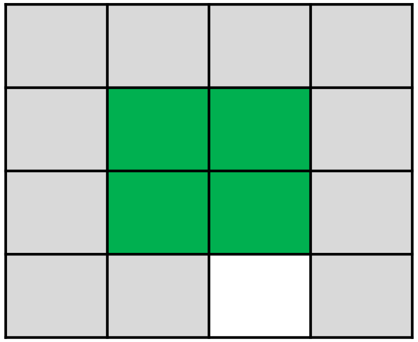
Backtracking în plan



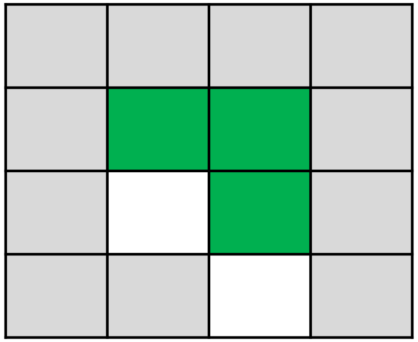
Backtracking în plan



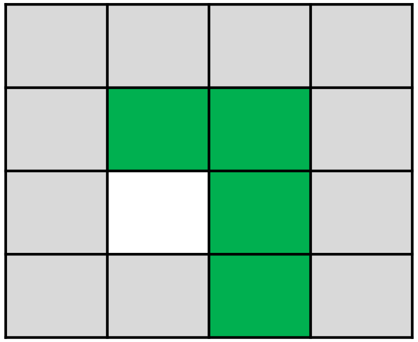
Backtracking în plan



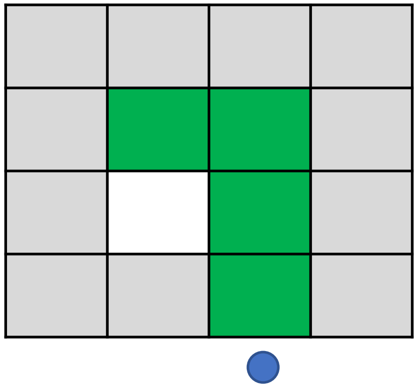
Backtracking în plan



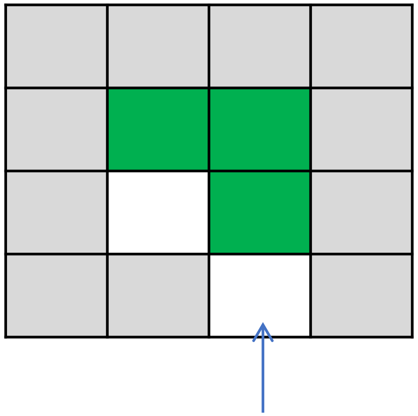
Backtracking în plan



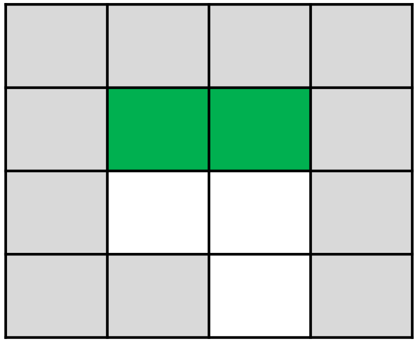
Backtracking în plan



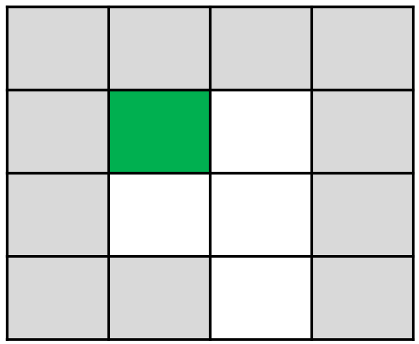
Backtracking în plan



Backtracking în plan



Backtracking în plan



Backtracking în plan

- Bordăm matricea cu 2 pentru a nu studia separat ieşirea din matrice.

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, unde

  $\mathbf{x}_i =$ a i-a celulă din drum

- **Condiţii interne (finale)**

- **Condiţii de continuare**

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, unde

  $\mathbf{x}_i =$ a i-a celulă din drum

- **Condiţii interne (finale)**

  $\mathbf{x}_k =$ celulă din afara matricei (marcată cu 2)

  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{k-1}\}$ – celule libere (marcată cu 0)

- **Condiţii de continuare**

  $\mathbf{x}_k$ celulă liberă **prin care nu am mai trecut**

Backtracking în plan

- **dacă poziţia este liberă şi putem continua, setăm a_{ij}=-1** (a fost **atinsă**), continuăm

- repunem a_{ij}=0 la întoarcere (din recursivitate)

Backtracking în plan

- **dacă poziţia este liberă şi putem continua, setăm a_{ij}=-1** (a fost **atinsă**), continuăm
- repunem a_{ij}=0 la întoarcere (din recursivitate)
- Matricea deplasărilor `depl` cu două linii şi `ndepl` coloane :

$$\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

```
void back(i, j){
   for (t = 1; t<=ndepl; t++){
      ii = i + depl[1][t]
      jj = j + depl[2][t];










   }
}
```

```
void back(i, j){
   for (t = 1; t<=ndepl; t++){
      ii = i + depl[1][t]
      jj = j + depl[2][t];
      if (a[ii][jj] == 1)
      else
         if (a[ii][jj] == 2)
            retsol(x,k);
        else
            if (a[ii][jj] == 0){









            }
   }
}
```

```
void back(i, j){
   for (t = 1; t<=ndepl; t++){
     ii = i + depl[1][t]
     jj = j + depl[2][t];
     if (a[ii][jj] == 1)
     else
        if (a[ii][jj] == 2)
          retsol(x,k);
       else
          if (a[ii][jj] == 0){
              k = k+1;        //creste
              xk ← (ii, jj);
              a[i][j] = -1; //marcam
              back(ii, jj);



          }
   }
}
```

```
void back(i, j){
   for (t = 1; t<=ndepl; t++){
      ii = i + depl[1][t]
      jj = j + depl[2][t];
      if (a[ii][jj] == 1)
      else
         if (a[ii][jj] == 2)
            retsol(x,k);
        else
            if (a[ii][jj] == 0){
                k = k+1;         //creste
                x_k ← (ii, jj);
                a[i][j] = -1; //marcam
                back(ii, jj);
                a[i][j] = 0;  //demarcam
                k = k-1 ;       //scade
            }
   }
}
```

Apel:

```
x₁ ← (i0, j0);
k = 1;
back(i0, j0)
```

Backtracking în plan

- **Cuvinte**. Se consideră un caroiaj (matrice) A cu m linii şi n coloane cu litere şi un cuvânt c.

  Să se determine dacă c se poate regăsi în matrice pornind dintr-o celulă şi deplasându-ne în oricare din celulele vecine pe orizontală, verticală sau diagonală fără a trece de două ori prin aceeaşi celulă – **TEMĂ**

  c = test

s	s	e	**s**	**t**	t	a
b	a	s	e	**e**	e	t
b	t	e	e	**t**	e	e
a	a	a	e	c	e	e

s	s	e	s	t	t	a
b	a	s	e	e	e	t
b	t	e	e	**t**	e	e
a	a	a	e	c	e	e

Backtracking în plan

- **Cuvinte**

Indicaţii:

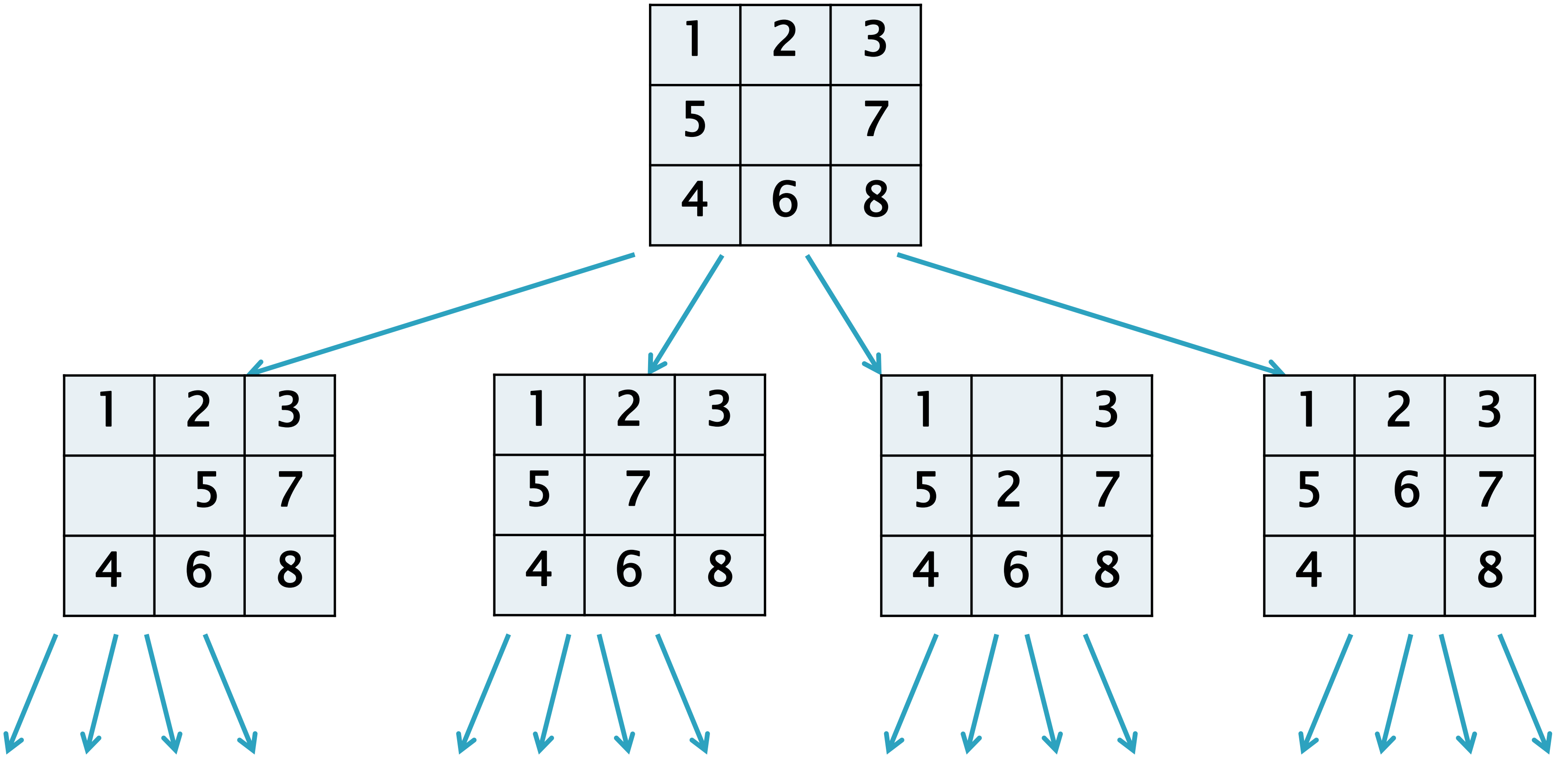
- similar cu Labirint
- punct de start poate fi orice celulă care conţine prima literă din c
- printre condiţii de continuare: litera la care am ajuns în matrice la pasul k trebuie să fie a k-a literă din c

Backtracking în plan

- **Arbori asociaţi mutărilor într-un joc**
  - Pentru dimensiuni mici
  - În multe cazuri arborele poate deveni de dimensiune mare şi un algoritm backtracking va fi lent
  - **Trebuie alte strategii de generare şi parcurgere a arborelui decât parcurgerea în adâncime**

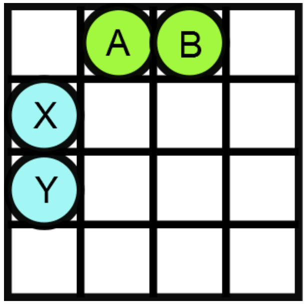
Backtracking în plan

- **Arbori asociaţi mutărilor într-un joc**
  - Pentru dimensiuni mici (exemplu: Perspico, Peg solitaire) https://en.wikipedia.org/wiki/Peg_solitaire
  - În multe cazuri arborele poate deveni de dimensiune mare şi un algoritm backtracking va fi lent
  - **Trebuie alte strategii de generare şi parcurgere a arborelui decât parcurgerea în adâncime**



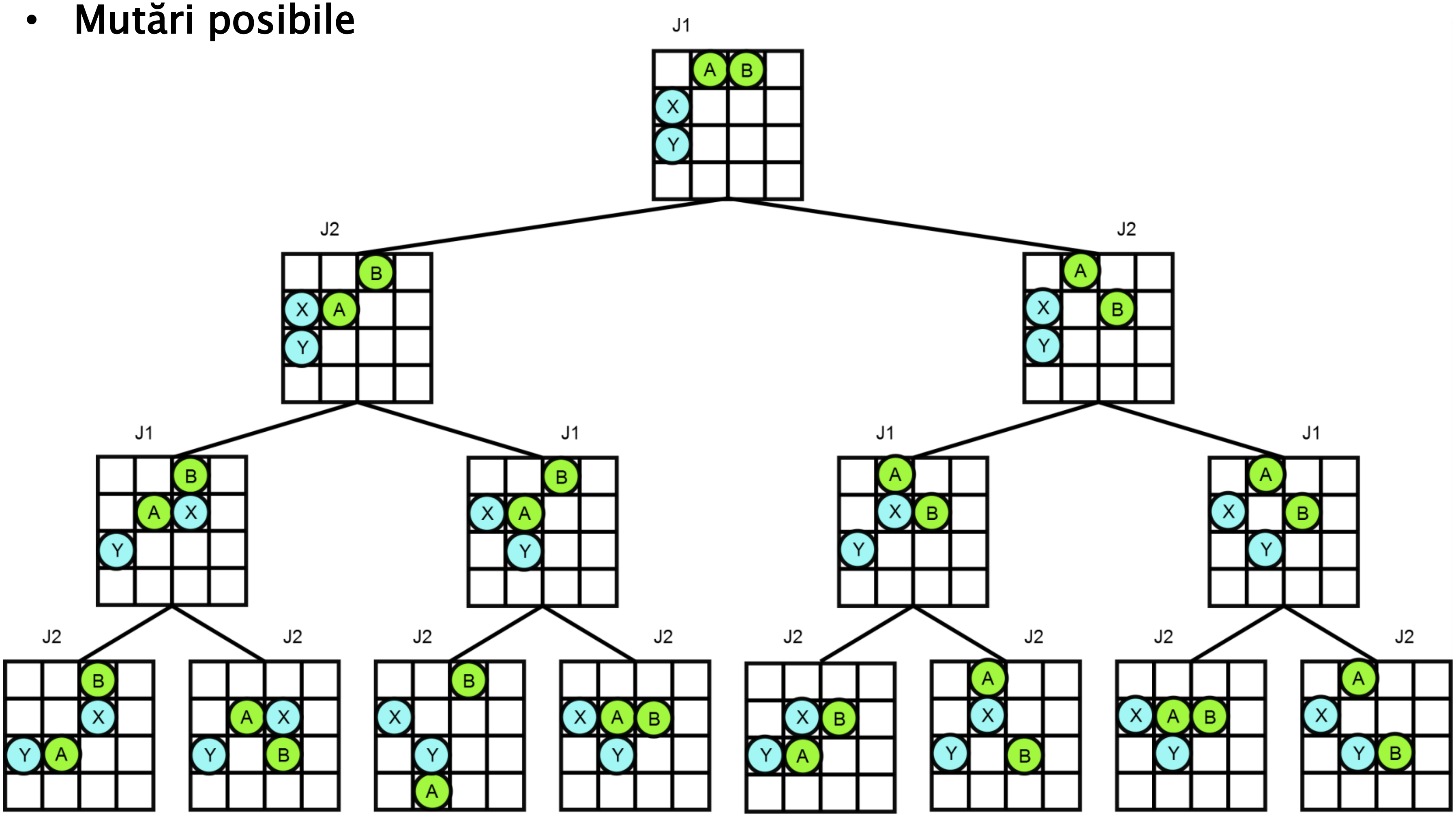
Perspico 3x3

- **2x2** fake-sugar-packet game
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-backtracking.pdf
- **Stare iniţială – tablă nxn**
  - **Jucatorul 1 – piese pe prima linie (A,B)**
    – mută piese în jos
    – trebuie să aducă piesele pe ultima linie
  - **Jucatorul 1 – piese pe prima coloană (X,)**
    – mută piese în dreapta
    – trebuie să aducă piesele pe ultima coloană



- **Câştigă – primul care aduce toate piesele unde trebuie**
- **Mutări posibile**
  – piesa în celula următoare (dacă e liberă)
  – piesa se mută 2 celule dacă sare o piesă a adversarului şi celula aflată la distanţa 2 este liberă

- **Mutări posibile**



2x2 fake-sugar-packet game
http://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-backtracking.pdf

Metoda Branch and Bound

Metoda Branch and Bound

Asemănări cu backtracking

- Pentru o configuraţie se poate **estima** dacă **nu** poate fi completată până la o soluție (mai bună decât cea mai bună soluţie determinată până la momentul curent, în cazul problemelor de optim) -> nu mai este explorată

nu se mai parcurge subarborele/ramificaţiile care il au ca rădăcină

⇒ branch and bound

Metoda Branch and Bound

Asemănări cu backtracking

- se aplică problemelor care pot fi reprezentate pe un arbore – la un pas avem de ales între mai multe variante

- Vârfurile arborelui (configuraţiile) corespund stărilor posibile în dezvoltarea soluţiei (soluţii parţiale)

- Pentru o configuraţie se poate **estima** dacă **nu** poate fi completată până la o soluție (mai bună decât cea mai bună soluţie determinată până la momentul curent, în cazul problemelor de optim)

Metoda Branch and Bound

- **Diferențe față de backtracking**
  - ordinea de parcurgere a arborelui (nu neapărat DF)
  - modul în care sunt eliminaţi subarborii care nu pot conduce la o soluţie
  - arborele poate fi infinit
  - util în probleme de optim

Metoda Branch and Bound

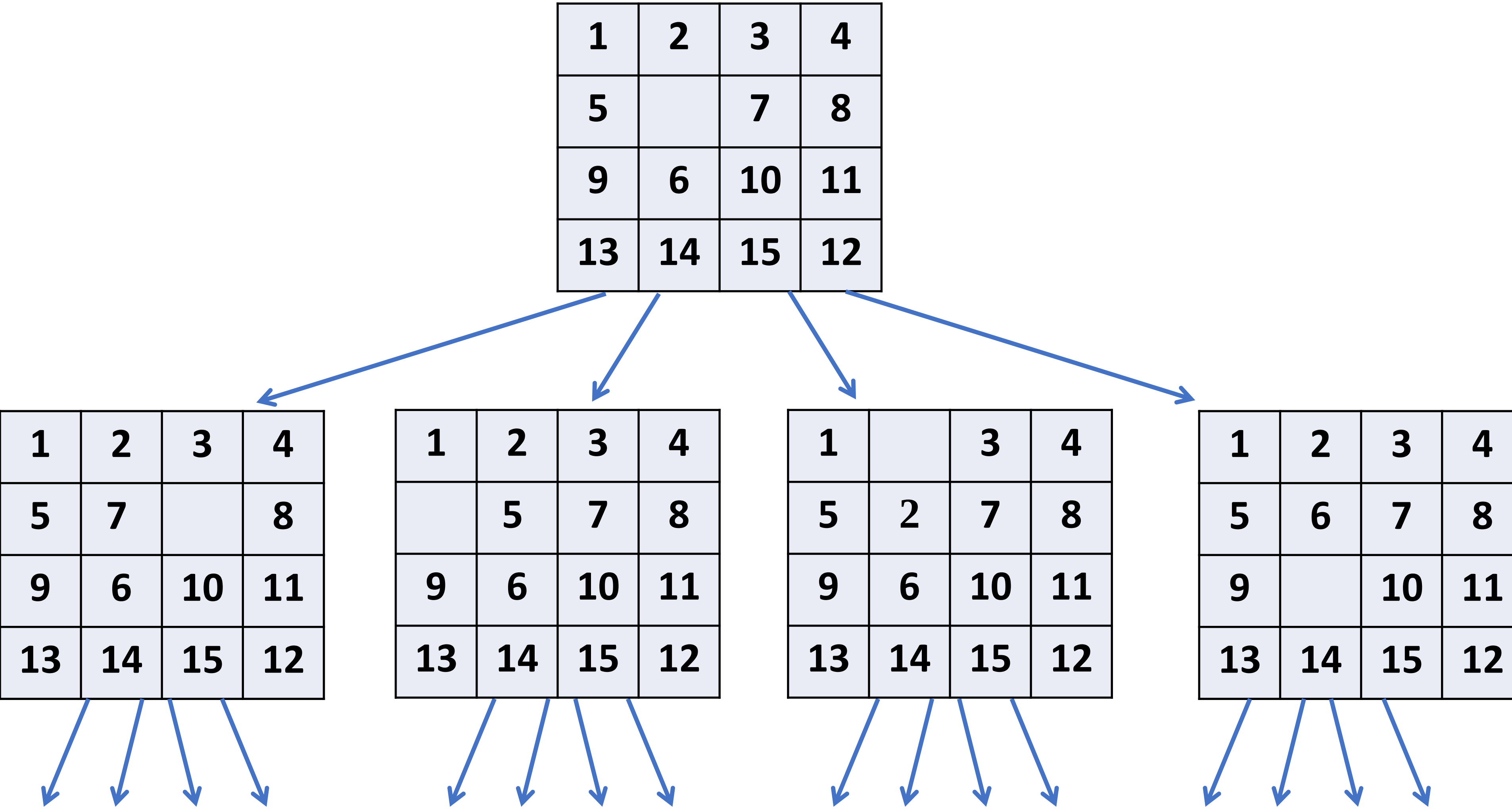
- Două tipuri de probleme la care se poate utiliza:
  - Se caută o anumită soluţie = un anumit ***vârf rezultat*** (final, frunză)
  - Se caută o **<u>soluție optimă</u>** (există mai multe vârfuri finale)
    - **probleme de optim – principalele aplicaţii**

Exemplu 1 - Jocul 15 (Perspico)

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

→

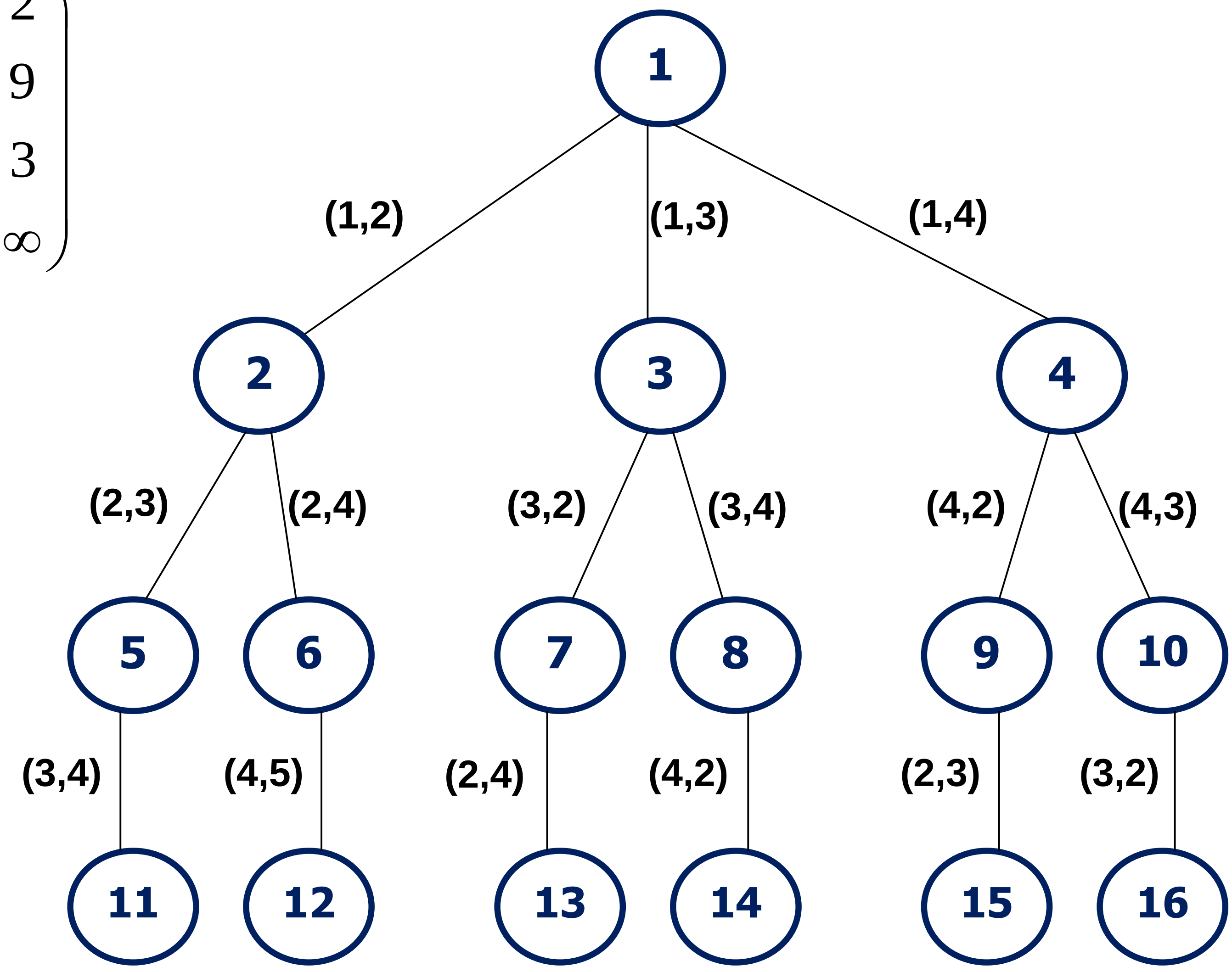
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	



1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 3 4
5 2 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12

Exemplul 2 – cirucit hamiltonian minim

$$C = \begin{pmatrix} \infty & 3 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



Metoda Branch and Bound

L –lisa de vârfuri **active** din arbore (care mai pot fi explorate)

Schemă

- Se inserează în L vârful inițial
- **Repetă**
    - Se **alege** un vârf din L care devine **curent**
    - Se generează fiii săi, care se adaugă în L

  până când vârful curent este final

Metoda Branch and Bound

Cum se alege vârful curent?

Metoda Branch and Bound

Cum se alege vârful curent

- **DF**

  - o parte arborele poate fi infinit

  - soluţia căutată poate fi de exemplu un fiu al rădăcinii diferit de primul fiu

Metoda Branch and Bound

Cum se alege vârful curent

- **DF**

  - o parte arborele poate fi infinit

  - soluţia căutată poate fi de exemplu un fiu al rădăcinii diferit de primul fiu

- **BF**

  - conduce totdeauna la soluţie, dar poate fi ineficientă dacă vârfurile au mulţi fii

Metoda Branch and Bound

- **Compromis**:

    Vârf x – asociat un **cost** pozitiv c(x)

    = măsură a gradului de "apropiere" a vârfului de o soluţie

Metoda Branch and Bound

- **Compromis**:

Vârf x – asociat un **cost** pozitiv c(x)

= măsură a gradului de "apropiere" a vârfului de o soluţie

➢ Este ales vârful de cost minim (! care **nu este neapărat fiu al vârfului curent anterior**)

➢ c(varf) ≤ c(fiu)

L va fi în general un min-ansamblu

Metoda Branch and Bound

- **Funcție de cost ideală**

  c(x) =

  - **nivel**(x), dacă x este vârf **rezultat**
  - +∞, dacă x este vârf **final**, diferit de vârf rezultat
  - **min** {c(y) | y fiu al lui x }, dacă x nu este vârf final

Metoda Branch and Bound

- **Funcție de cost ideală**

  c(x) =

  - **nivel**(x), dacă x este vârf **rezultat**
  - +∞, dacă x este vârf **final**, diferit de vârf rezultat
  - **min** {c(y) | y fiu al lui x }, dacă x nu este vârf final

    ➢ **Nu se poate calcula pentru arbore infinit**

    ➢ **Trebuie parcurs arborele în întregime pentru a o calcula**

Metoda Branch and Bound

- **Aproximație f a lui c**

Metoda Branch and Bound

- **Aproximație f a lui c**
  - f(x) să poată fi calculată doar pe baza informaţilor din drumul de la rădăcină la x
  - este indicat ca **f ≤ c** (aproximare optimistă)

Metoda Branch and Bound

- **Condiţie compromis DF/BF**:

Există un k natural - pentru orice vârf `x` situat pe un nivel `nx` şi orice vârf `y` situat pe un nivel $ny \geq nx+k$, să avem

$$f(x) < f(y)$$

$\Rightarrow$ **dacă există soluție, ea va fi atinsă într-un timp finit**

Metoda Branch and Bound

- **Cum definim f?**

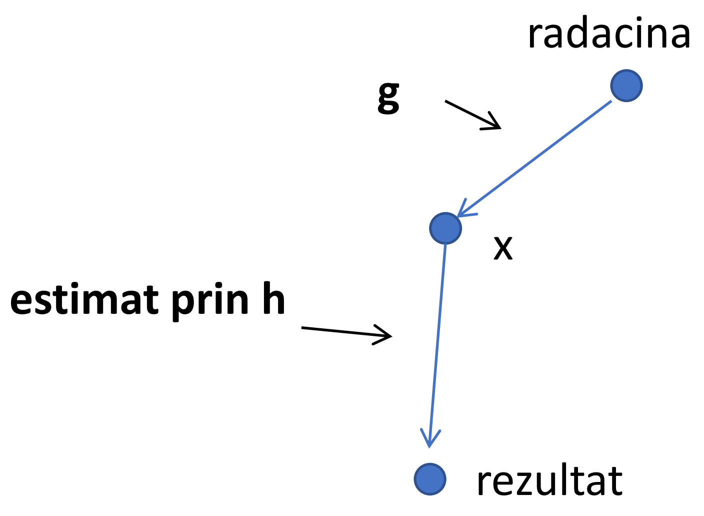
Metoda Branch and Bound

- **Cum definim f? O posibililitate ar fi:**

$$\mathtt{f(x)} = \mathtt{g(x)+h(x)}$$

❑ **`g(x)`** = distanţa de la rădăcină la nodul $\mathtt{x}$

❑ **`h(x)`** = estimarea distanţei de la $\mathtt{x}$ la vârful rezultat ("**subestimare**")



Exemplul 1 - Jocul 15 (Perspico)

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

→

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Exemplul 1 - Jocul 15 (Perspico)

- **Condiție de existență**

  - căsuța liberă **16**, pe poziția (l, c)

  - pentru plăcuța etichetată cu i calculăm

n(i) = numărul locaşurilor care îi urmează şi care conţin o plăcuţă cu etichetă mai mică decât i

Există soluție ⇔

n(1)+n(2)+...+n(16)+(l+c) %2 este par.

Exemplul 1 - Jocul 15 (Perspico)

- Putem considera euristici precum:
  - h(t) = numărul de plăcuţe care nu sunt la locul lor
  - h(t) = **distanţa Manhattan** = suma pentru fiecare căsuță a **numărului de mutări** pentru a o putea aduce în poziţia finală
    - dacă o cifră este pe poziţia (i,j) şi trebuie adusă pe poziţia (r,s) distanţa este

$$|r-i|+|s-j|$$

Exemplul 1 - Jocul 15 (Perspico)

- Putem considera euristici precum:
  - h(t) = numărul de plăcuţe care nu sunt la locul lor
  - h(t) = **distanţa Manhattan** = suma pentru fiecare căsuță a **numărului de mutări** pentru a o putea aduce în poziţia finală

  - h(t) = numărul de plăcuţe care nu sunt la locul lor

h = câte plăcuţe nu sunt la locul lor

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

g = 0
h = 5

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

g=0,h=5

f = 5

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

g=1,h=6

f=7

1		3	4
5	2	7	8
9	6	10	11
13	14	15	12

g=1,h=6

f=7

1	2	3	4
	5	7	8
9	6	10	11
13	14	15	12

g=1,h=4

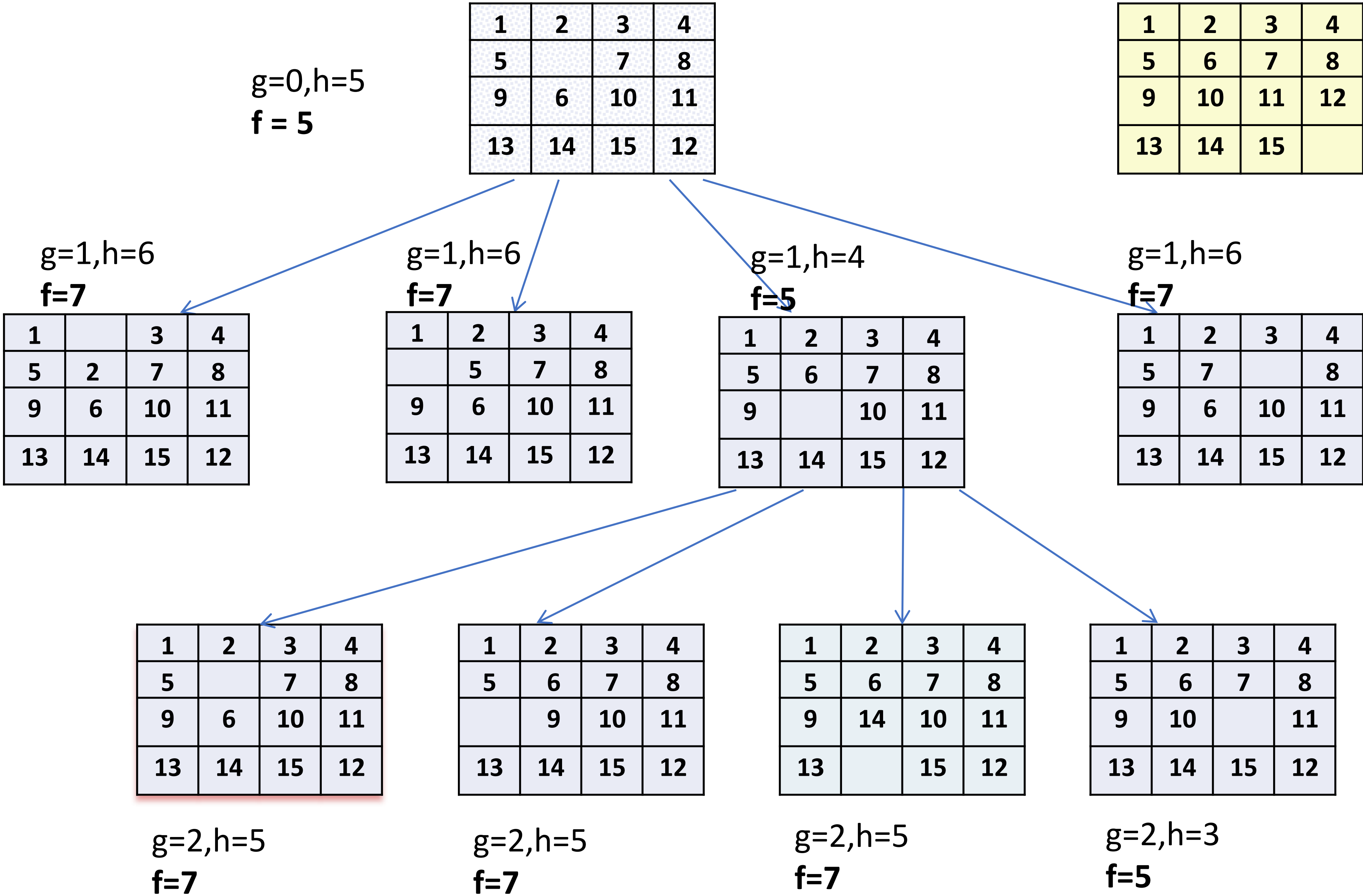
f=5

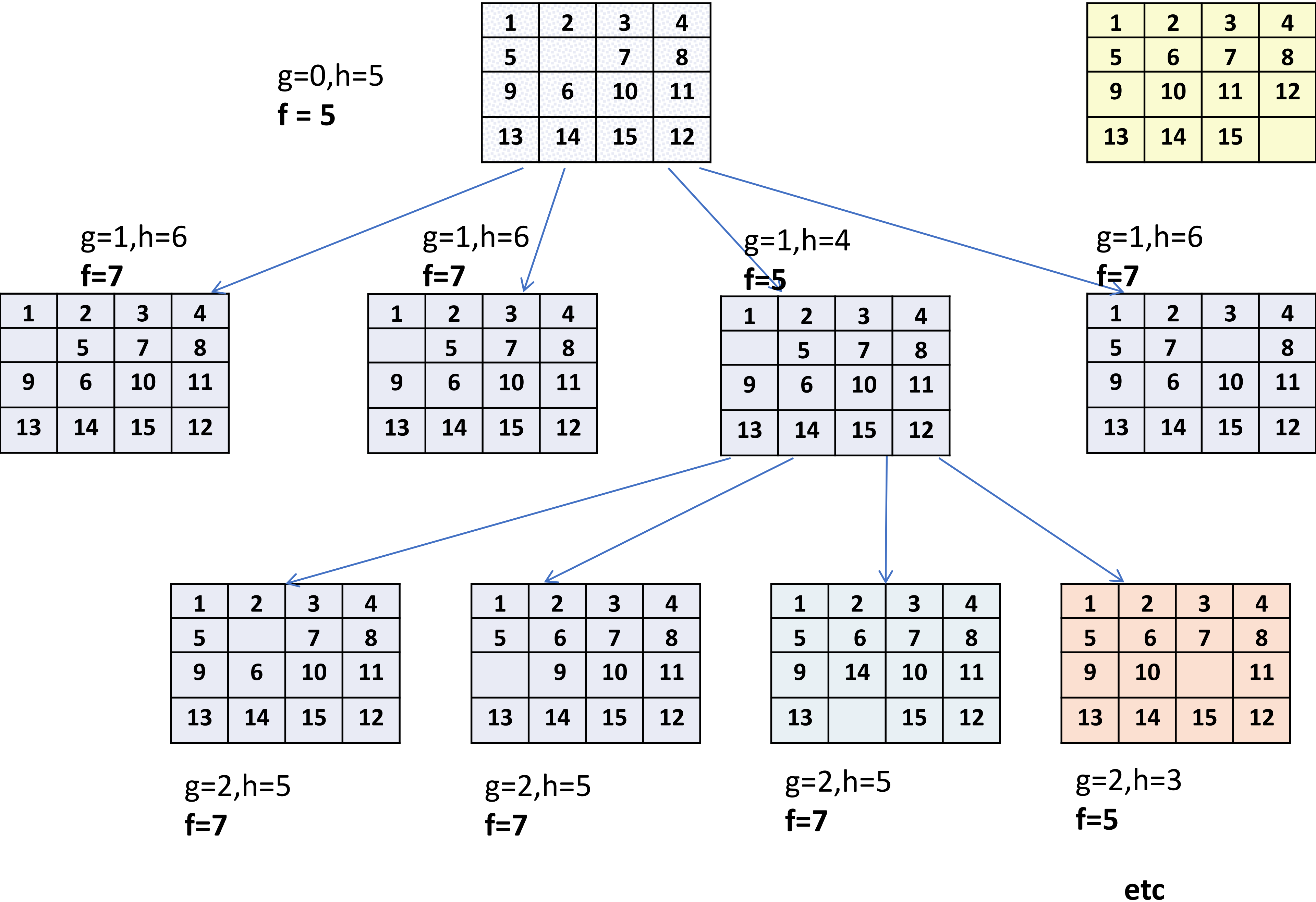
1	2	3	4
5	6	7	8
9		10	11
13	14	15	12

g=1,h=6

f=7

1	2	3	4
5	7		8
9	6	10	11
13	14	15	12


g=0,h=5
f = 5
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15
g=1,h=6
f=7
1 3 4
5 2 7 8
9 6 10 11
13 14 15 12
g=1,h=6
f=7
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=1,h=4
f=5
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
g=1,h=6
f=7
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 14 10 11
13 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
g=2,h=3
f=5



1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=0,h=5
f = 5
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15
g=1,h=6
f=7
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=1,h=6
f=7
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=1,h=4
f=5
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=1,h=6
f=7
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
1 2 3 4
5 7 8
9 6 10 11
13 14 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 14 10 11
13 15 12
g=2,h=5
f=7
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
g=2,h=3
f=5
etc

Metoda Branch and Bound

- **Pentru probleme de optim**

  - **Vom considera probleme de minimizare**

  - Maximizare – similar

    (maximizare obiectiv g -> minimizare obiectiv –g)

Metoda Branch and Bound

- **Pentru probleme de optim (minim)**
  - Considerăm **`lim`** - **aproximaţie prin adaos** a minimului căutat, care se actualizează pe parcursul algoritmului = costul celei mai bune soluţii găsite până la pasul curent

Metoda Branch and Bound

- **Pentru probleme de optim (minim)**
  - Configuraţiile cu costul estimat mai mare decât **`lim`** nu mai trebuie considerate:

    `cost real ≥ cost estimat  > lim`

  - Dacă o configuraţie finală (corespunzătoare unei soluţii posibile) are costul estimat mai mic decât **`lim`**, **`lim`** se actualizează şi sunt eliminate din lista vârfurilor active vârfurile cu cost mai mare decât noul lim

Metoda Branch and Bound

- **Pentru probleme de optim (minim)**
  - Notăm rad – vârful corespunzător configurației inițiale

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
```

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
while L ≠ ∅
    i ⇐ L {scos vârful i cu f(i) minim din min-ansamblul L}
    for toţi j fii ai lui i
```

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
while L ≠ ∅
   i ⇐ L {scos vârful i cu f(i) minim din min-ansamblul L}
   for toţi j fii ai lui i
      calculăm f(j); calcule locale asupra lui j;
     tata(j) ← i
```

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
while L ≠ ∅
    i ⇐ L {scos vârful i cu f(i) minim din min-ansamblul L}
    for toţi j fii ai lui i
        calculăm f(j); calcule locale asupra lui j;
      tata(j) ← i
        if j este vârf final
                if f(j)<min
                    min ← f(j); i_final ← j
                    elimină din L vârfurile k cu f(k)≥ min
```

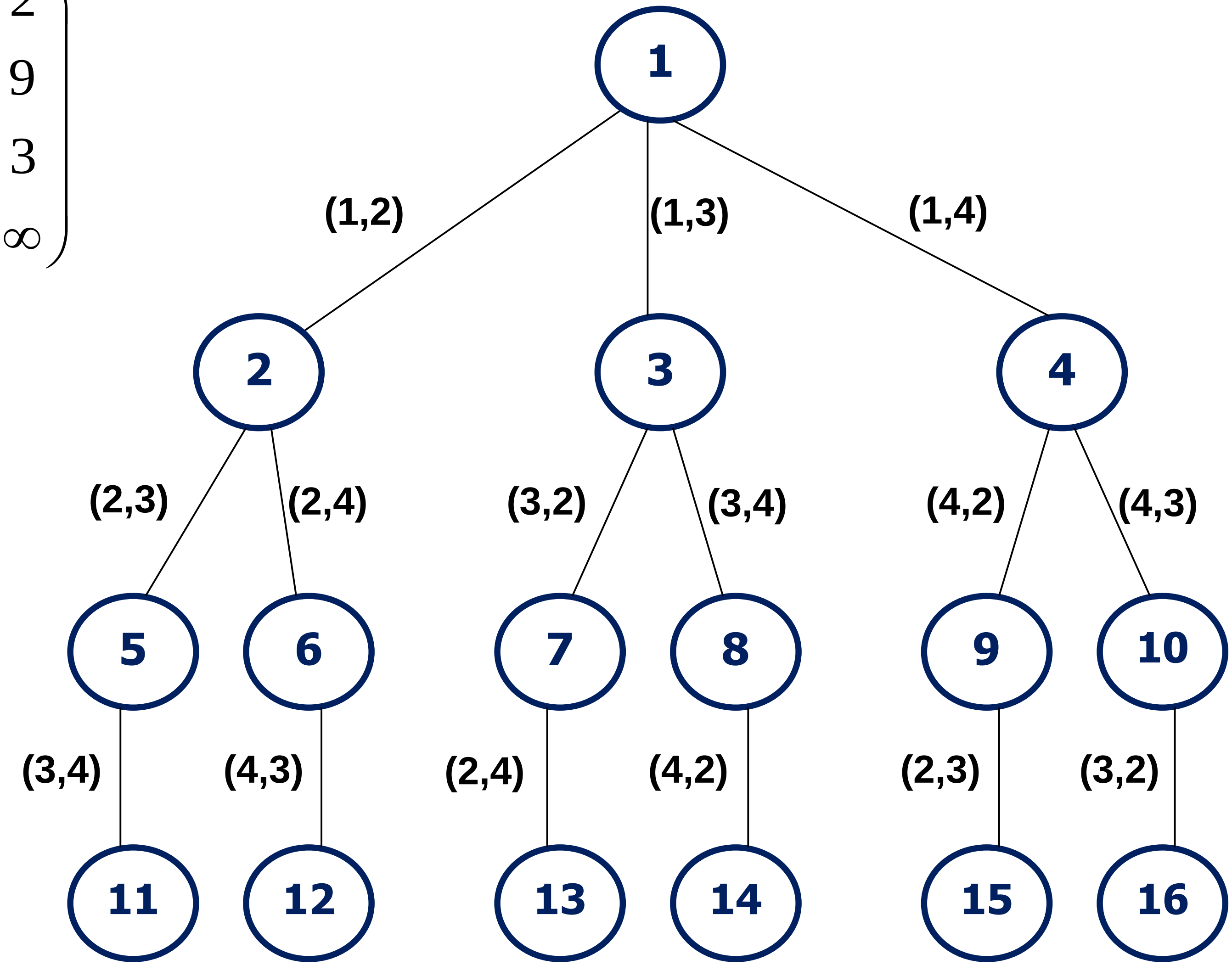
```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
while L ≠ ∅
    i ⇐ L {scos vârful i cu f(i) minim din min-ansamblul L}
    for toţi j fii ai lui i
        calculăm f(j); calcule locale asupra lui j;
      tata(j) ← i
        if j este vârf final
                if f(j)<min
                    min ← f(j); i_final ← j
                    elimină din L vârfurile k cu f(k)≥ min
      else
              if f(j)<min
                j ⇒ L
```

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm f(rad); tata(i) ← 0
while L ≠ ∅
   i ⇐ L {scos vârful i cu f(i) minim din min-ansamblul L}
   for toţi j fii ai lui i
      calculăm f(j); calcule locale asupra lui j;
     tata(j) ← i
      if j este vârf final
           if f(j)<min
              min ← f(j); i_final ← j
              elimină din L vârfurile k cu f(k)≥ min
     else
         if f(j)<min
           j ⇒ L

if min=lim
     write('Nu există soluţie')
else writeln(min); i ← i_final
      while i ≠ 0
          write(i); i ← tata(i)
```

Exemplul 2 – circuit hamiltonian minim TSP

$$C = \begin{pmatrix} \infty & 3 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



Exemplu – circuit hamiltonian minim TSP

Pentru un vârf x din arbore valoarea c(x) dată de funcţia de cost **ideală** este:

- lungimea circuitului corespunzător lui x dacă x este frunză
- min {c(y) | y fiu al lui x } altfel.

Exemplu – circuit hamiltonian minim TSP

Cum determinăm limita inferioară f(x) pentru circuitului minim corespunzător vârfului x din arborele Branch and Bound

Exemplu – circuit hamiltonian minim TSP

Cum determinăm limita inferioară f(x) pentru circuitului minim corespunzător vârfului x din arborele Branch and Bound

Idee simplă - lungimea circuitului minim care corespunde lui x este $\leq$ lungimea drumului de la rădăcină la x (arcelor alese pentru a ajunge la x)

`f(x) = g(x) + h(x)`

- **`g(x)`** = distanţa de la rădăcină la nodul x
- **`h(x)`** = 0 -> este o **"subestimare"**

Exemplu – circuit hamiltonian minim TSP

Cum determinăm limita inferioară f(x) pentru circuitului minim corespunzător vârfului x din arborele Branch and Bound

- `h(x)` = 0 -> este o **"subestimare“**
- **Problemă – cu cât estimarea este mai puţin precisă (mai depărtată de valoarea reală) sunt excluse mai puţine vârfuri din parcurgere ⇒ algoritm mai lent**

L = {1}

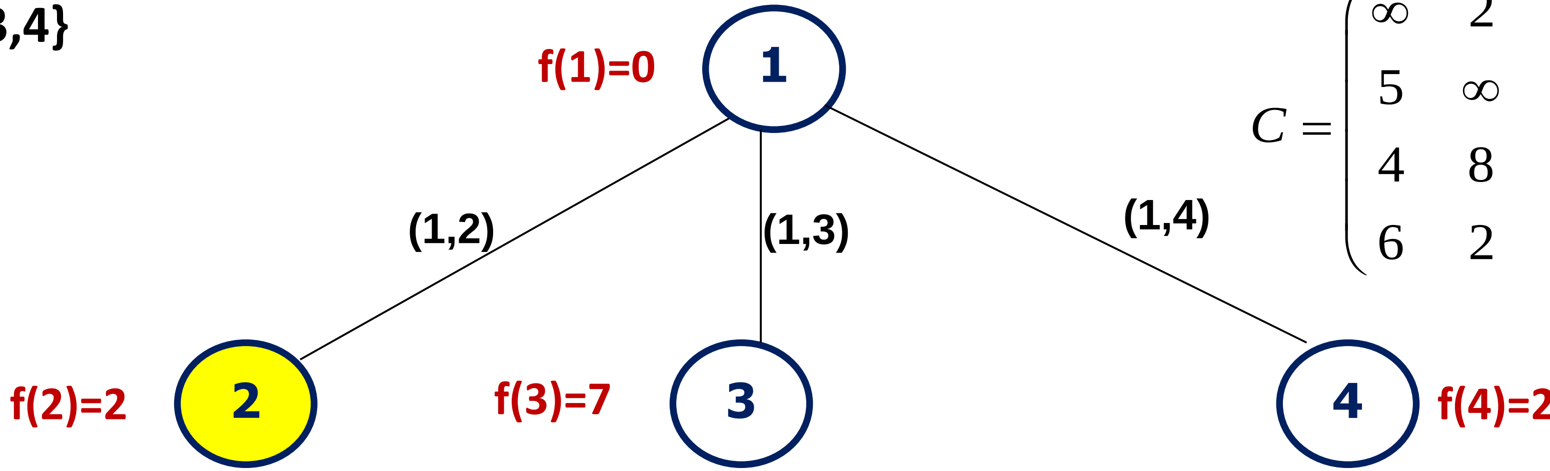
f(1)=0

1

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

L = {2,3,4}

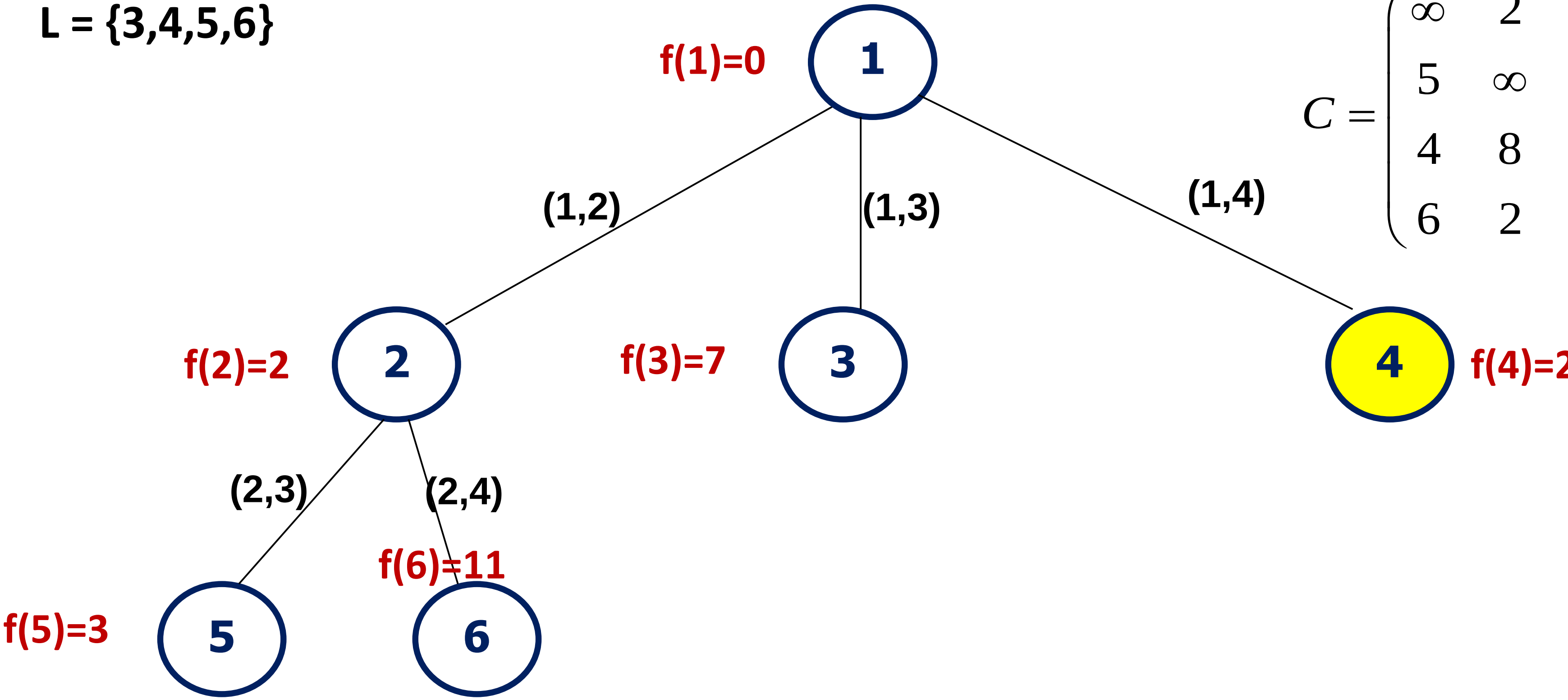
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



Extragem din L vârful cu f minim şi îi generăm fii

L = {3,4,5,6}

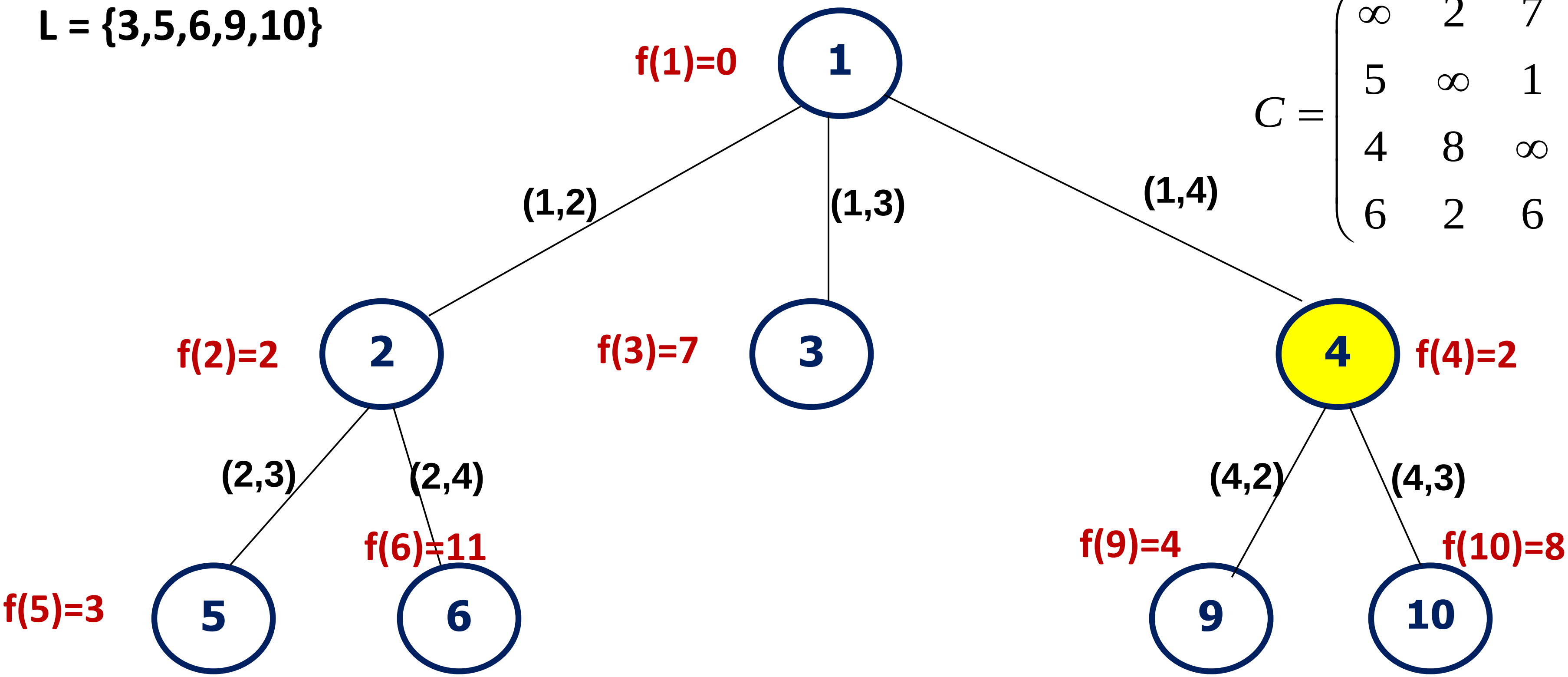
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



Extragem din L vârful cu f minim şi îi generăm fii

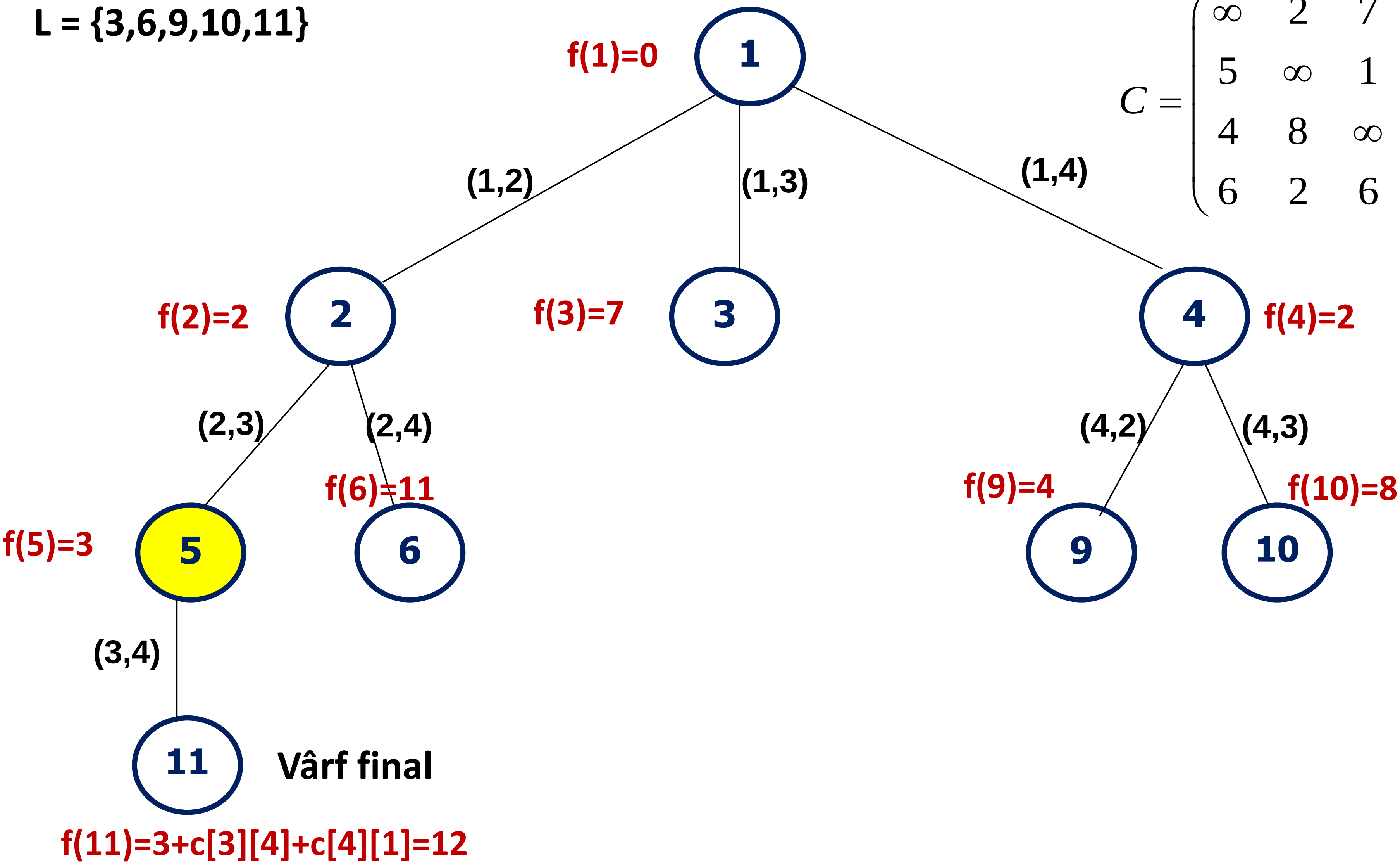
L = {3,5,6,9,10}

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



L = {3,6,9,10,11}

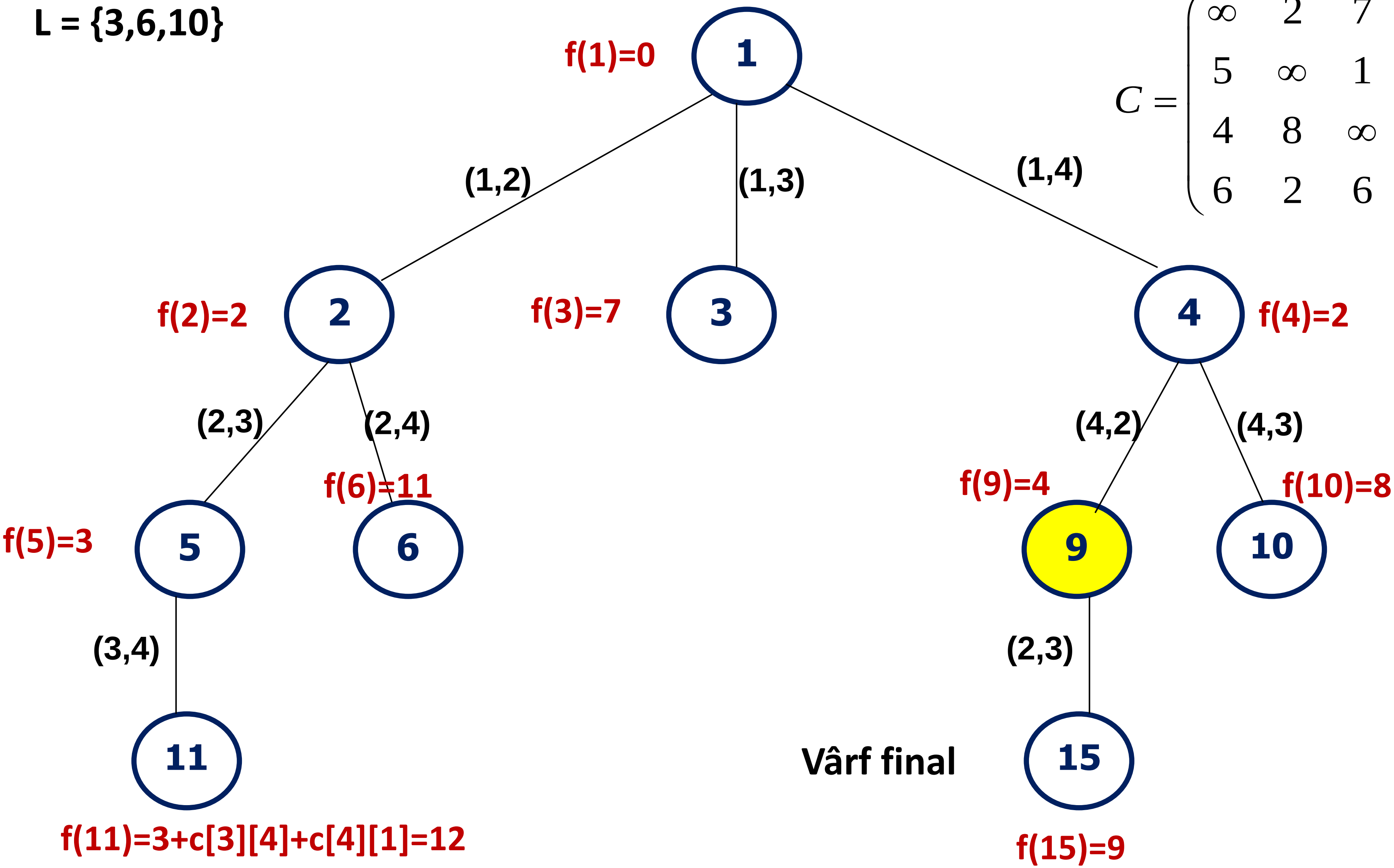
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim=12 - > eliminăm vârfurile cu f mai mare decât 12

L = {3,6,10}

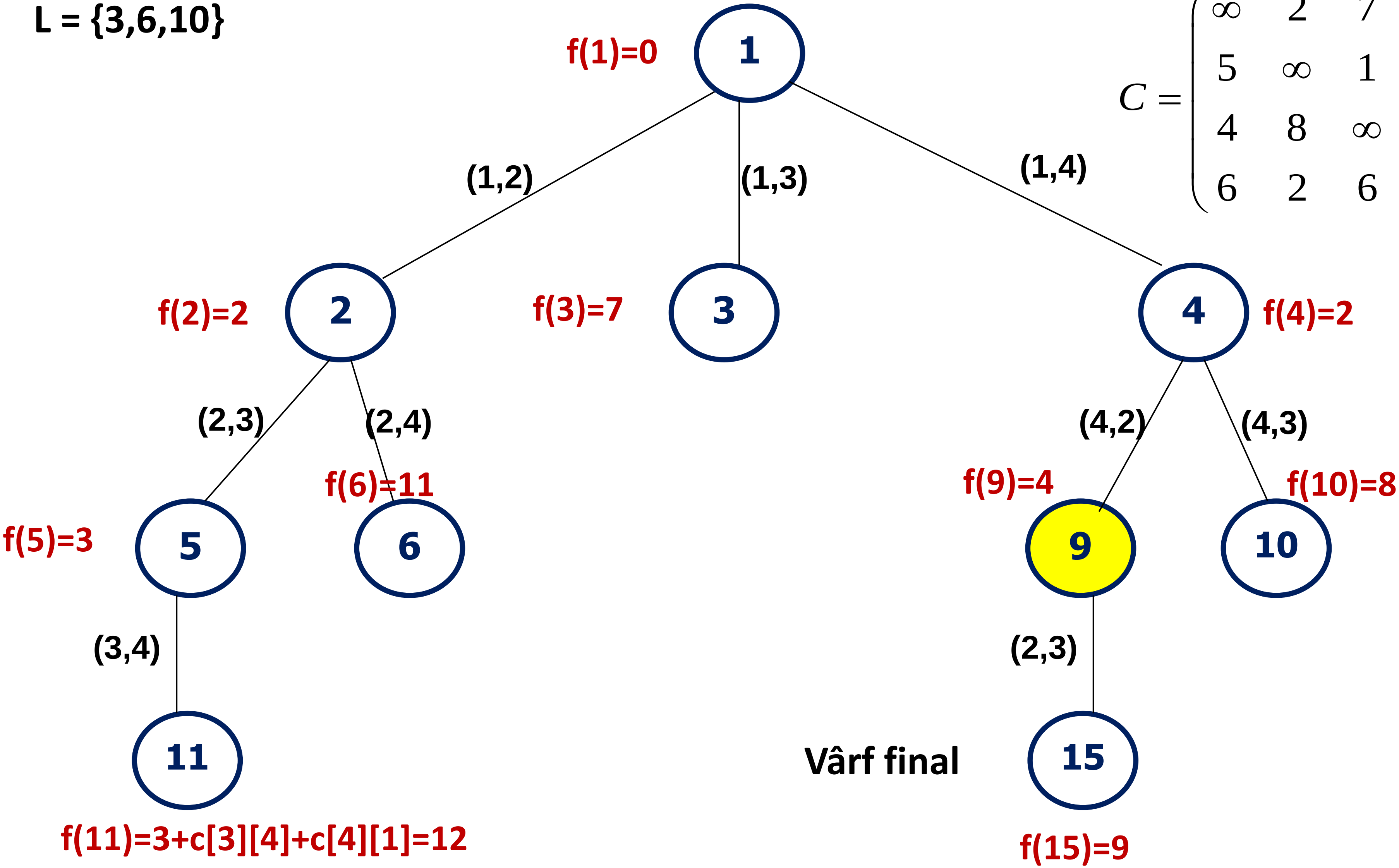
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim=min{12, 9} = 9 - > eliminăm vârfurile cu f mai mare decât 9

L = {3,6,10}

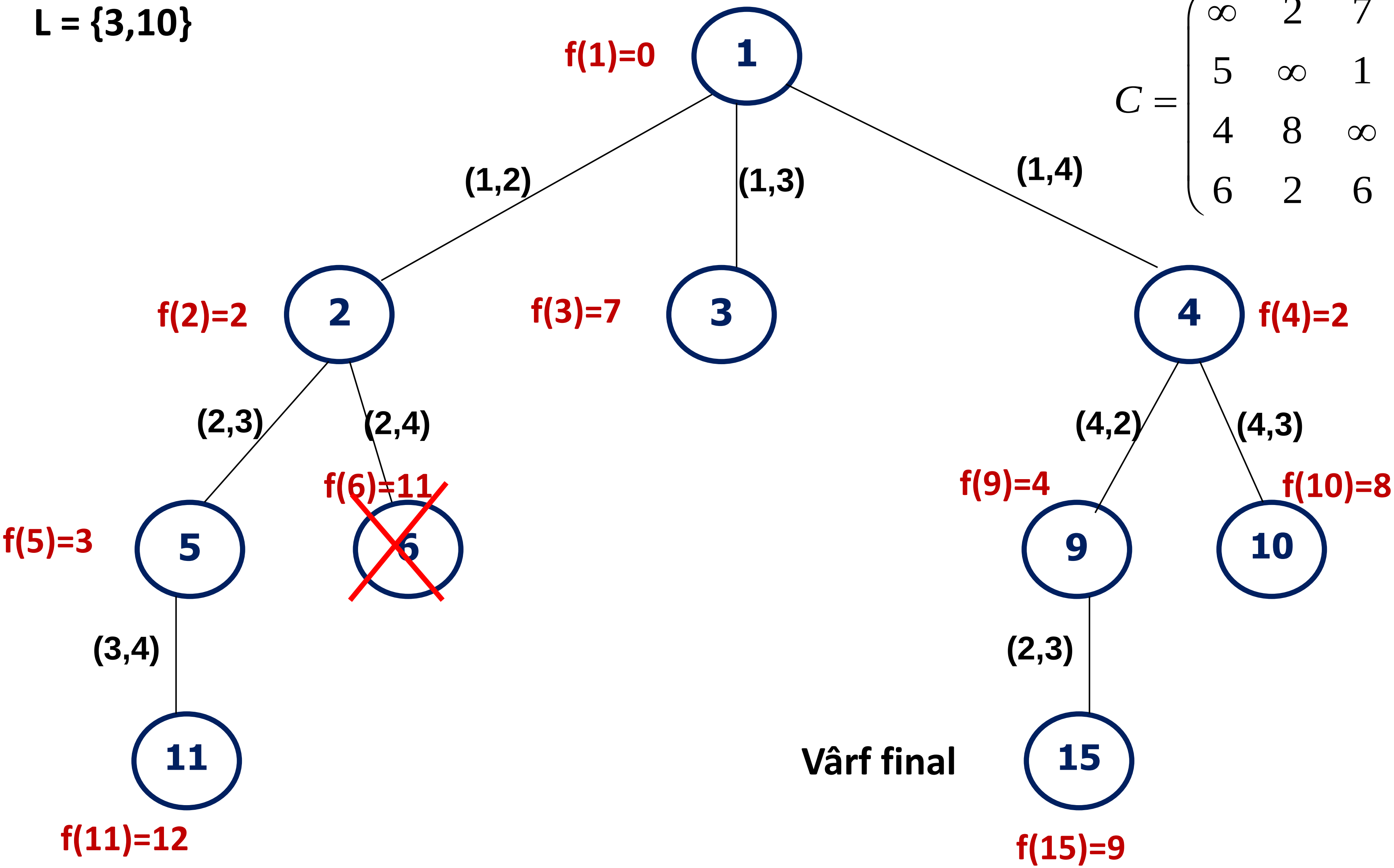
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim=min{12, 9} = 9 - > eliminăm vârfurile cu f mai mare decât 9

L = {3,10}

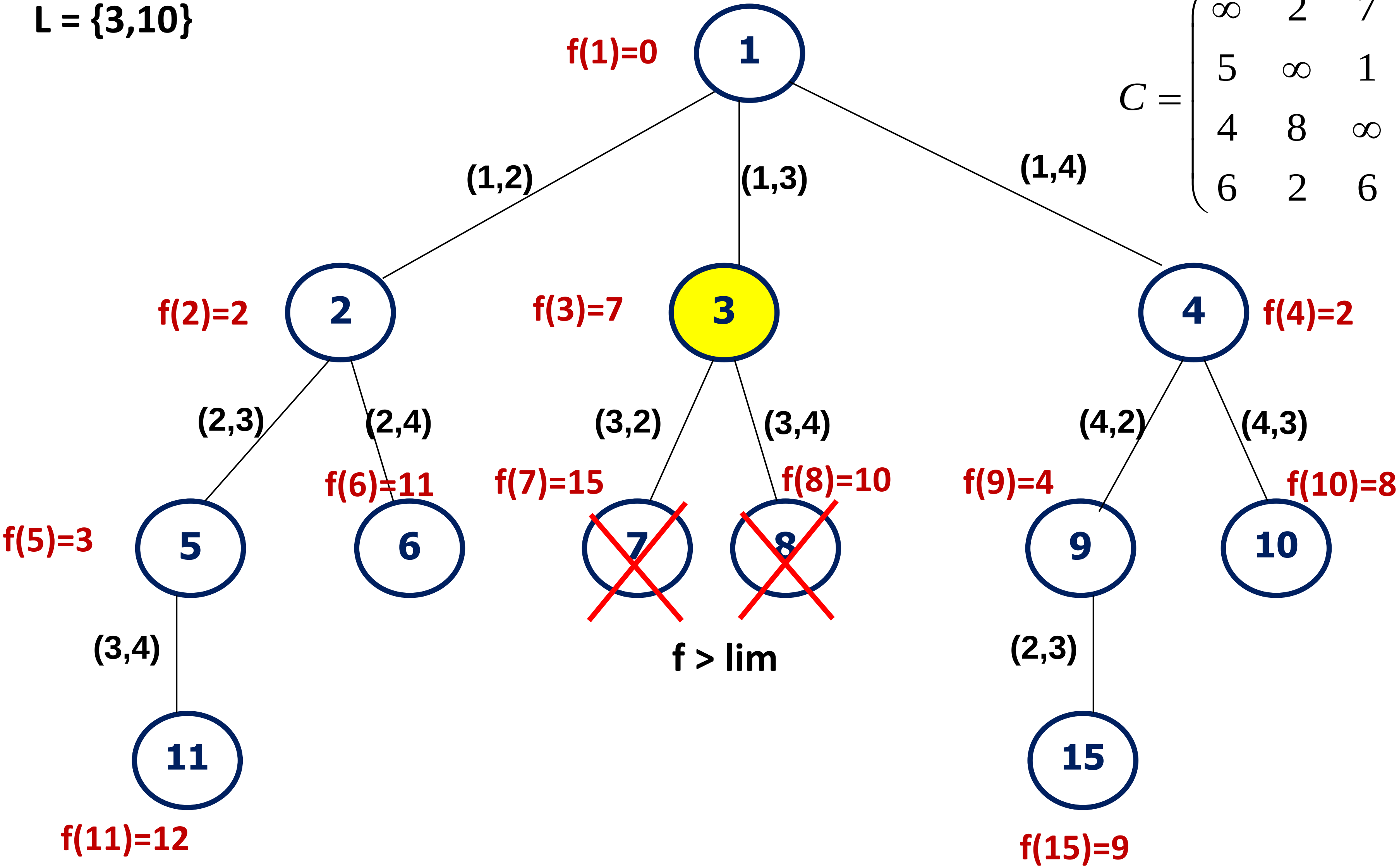
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim= 9

L = {3,10}

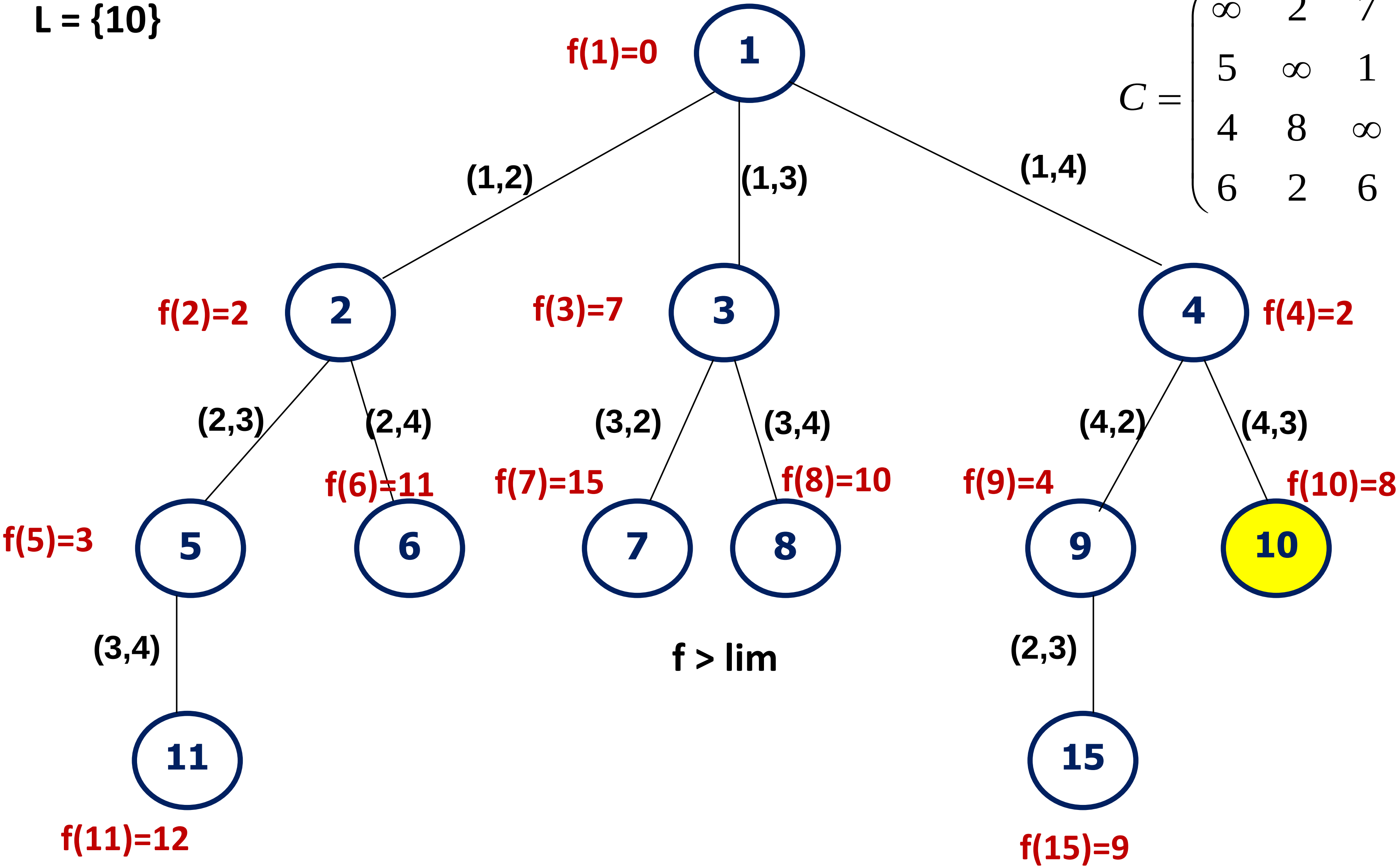
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim= 9

L = {10}

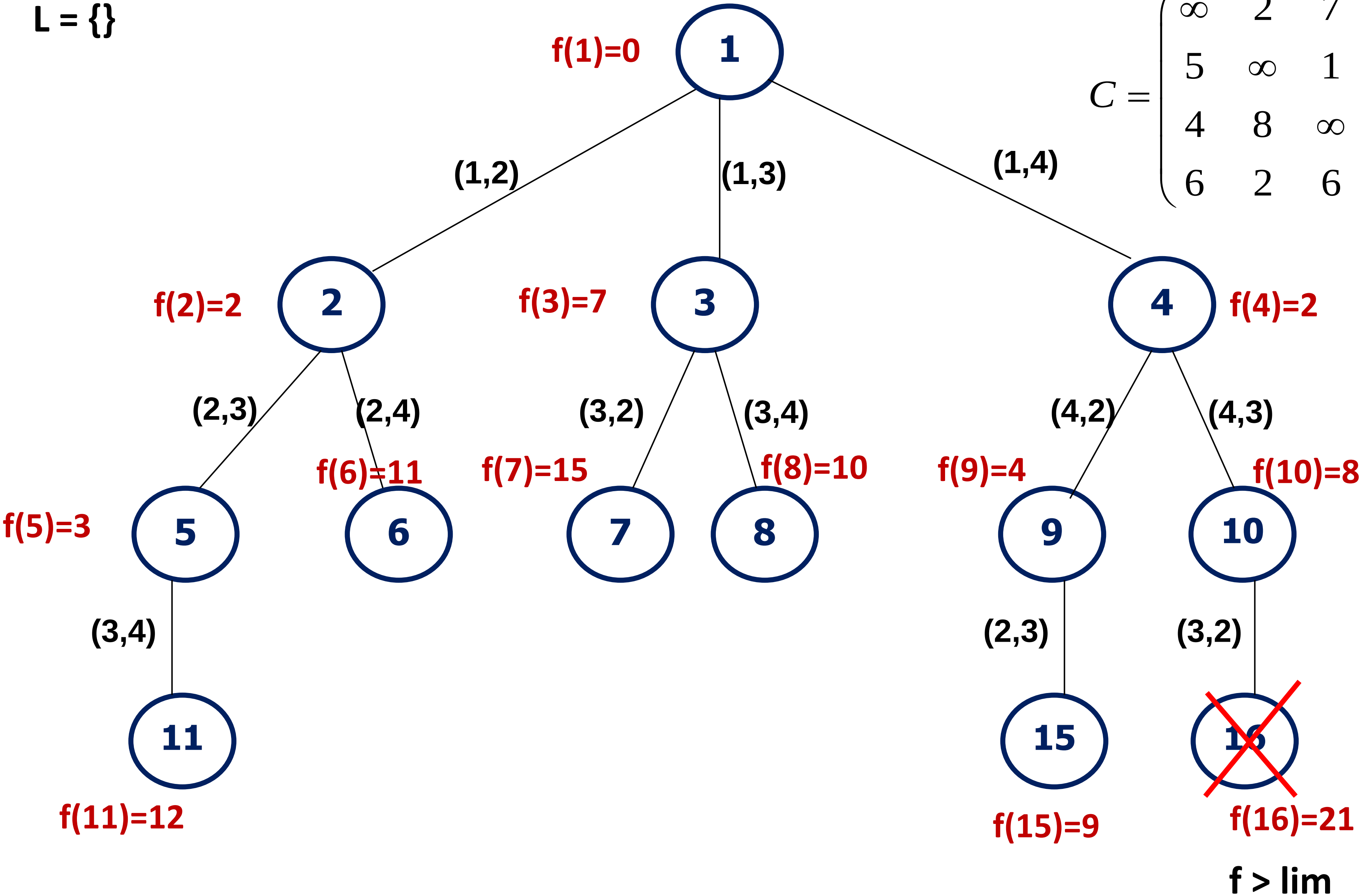
$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim= 9

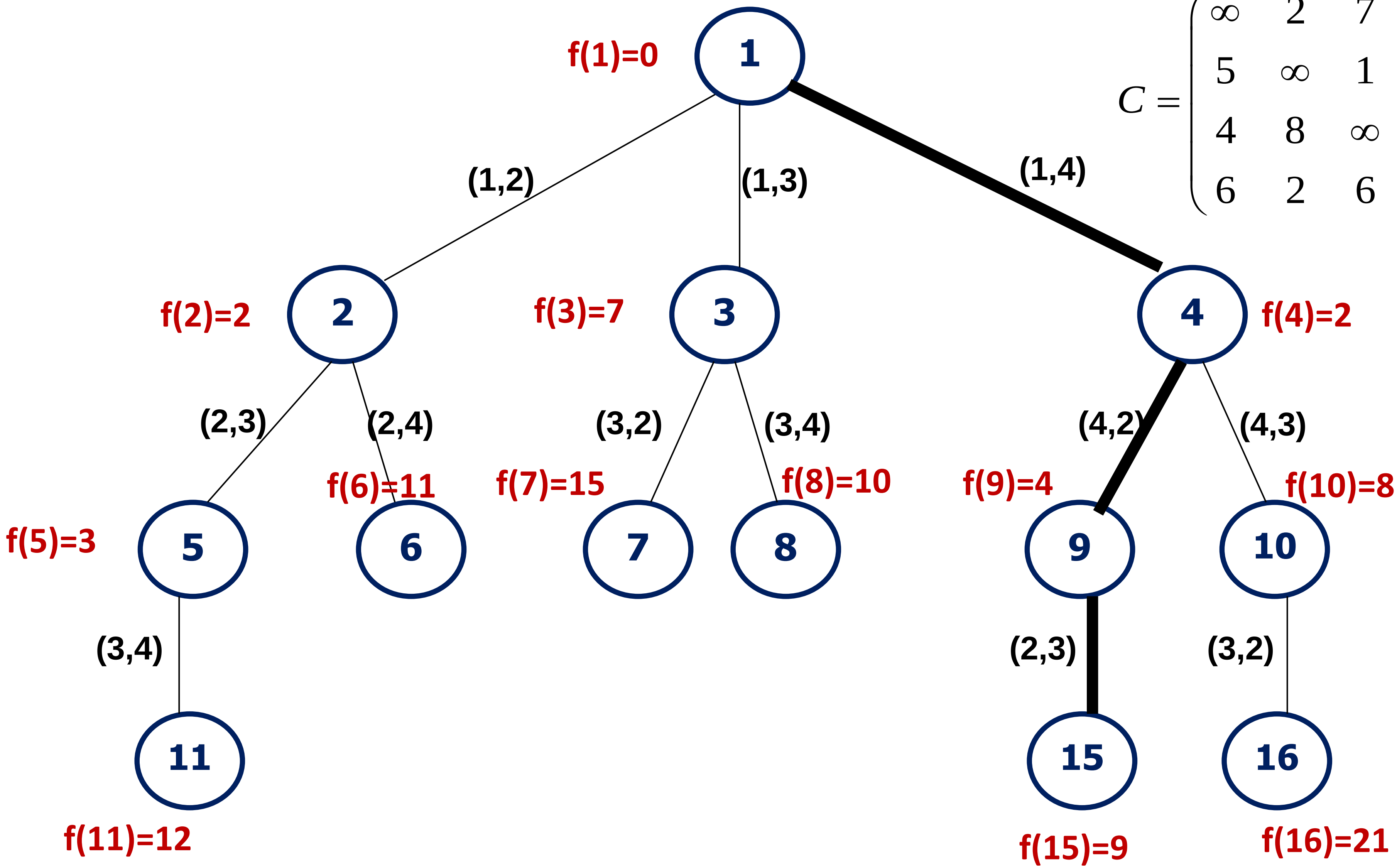
L = {}

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

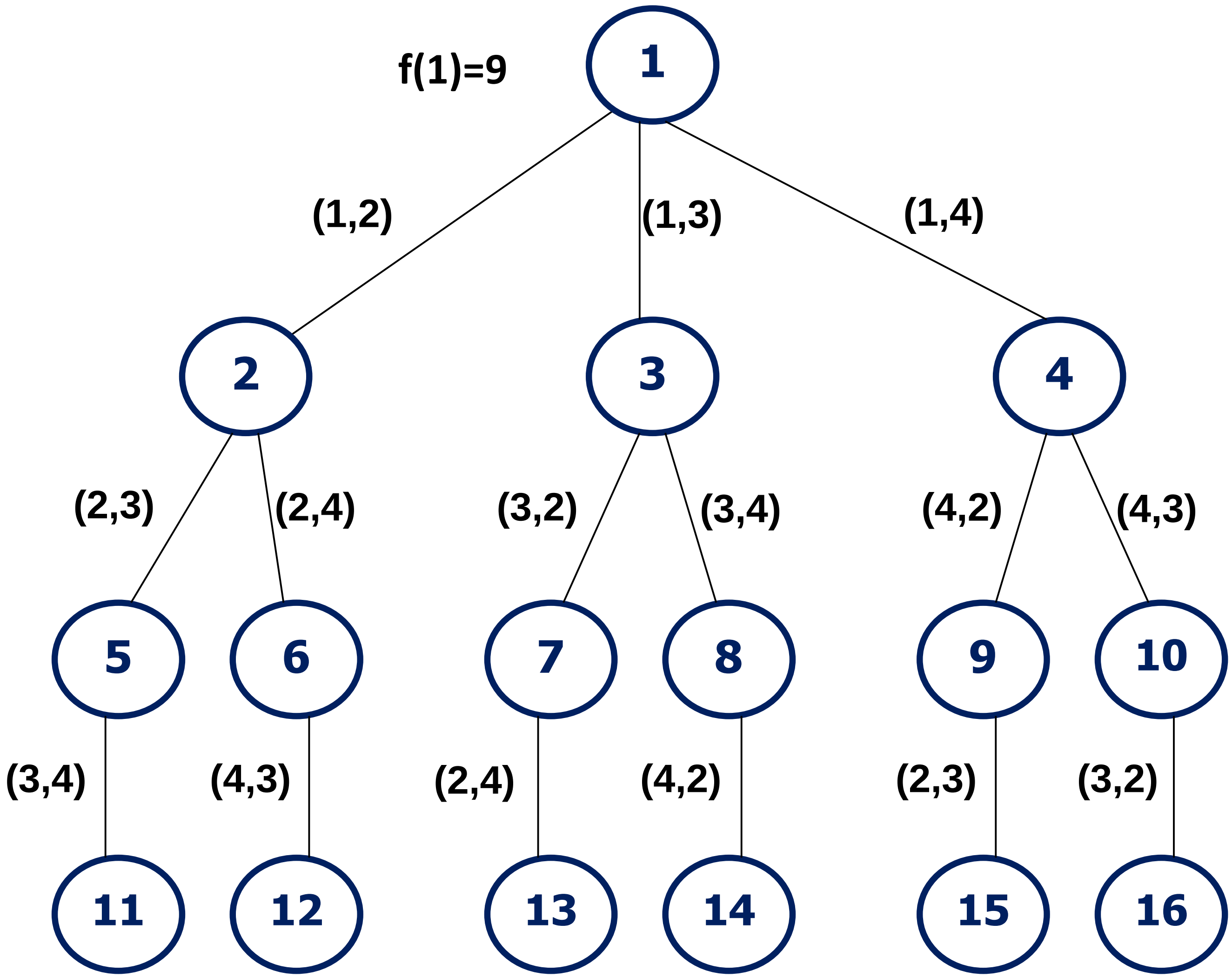


lim= 9

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$



lim= 9



Au fost excluse puţine vârfuri de la expandare

Exemplu – circuit hamiltonian minim TSP

Euristici h mai precise:

- `cost(TSP) ≥`

$$\left(\frac{1}{2}\sum_{v\in V}\text{arc minim care intra in v} + \sum_{v\in V}\text{arc minim care iese din v}\right)$$

- Pentru o configuraţie care deja conţine anumite arce - se actualizează limita inferioară

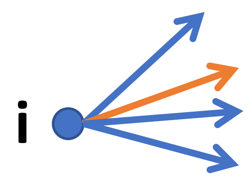
- Pentru gestionare – se modifică succesiv matricea costurilor

Circuit hamiltonian minim TSP

- **Observația 1**. Dacă micşorăm toate elementele unei linii i din matricea de costuri C cu α, orice circuit hamiltonian va avea costul micşorat cu α **De ce?**
- Vom lucra cu **matrice de costuri reduse** = în care pe orice linie sau coloană apare cel puţin un zero, exceptând cazul când linia sau coloana conţine numai ∞

Circuit hamiltonian minim TSP

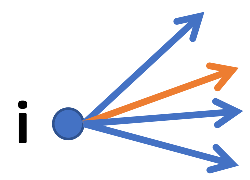
- **Observația 1**. Dacă micşorăm toate elementele unei linii i din matricea de costuri C cu α, orice circuit hamiltonian va avea costul micşorat cu α



Un circuit conţine un unic arc care iese din i

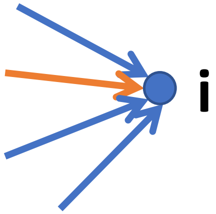
Circuit hamiltonian minim TSP

- **Observația 1**. Dacă micşorăm toate elementele unei linii i din matricea de costuri C cu α, orice circuit hamiltonian va avea costul micşorat cu α



Circuit hamiltonian minim TSP

- **Observația 2**. Dacă micşorăm toate elementele unei coloane din matricea de costuri C cu α, orice circuit hamiltonian va avea costul micşorat cu α



Circuit hamiltonian minim TSP

- **matrice de costuri reduse** = în care pe orice linie sau coloană apare cel puţin un zero, exceptând cazul când linia sau coloana conţine numai ∞

- f(rad) = cantitatea (totală) cu care se reduce
  matricea de cost C

  = limită inferioară pentru costul CHM

Circuit hamiltonian minim TSP

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

Reducem linia 1 cu 2

$$\begin{pmatrix} \infty & 0 & 5 & 0 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

Reducem linia 2 cu 1

$$\begin{pmatrix} \infty & 0 & 5 & 0 \\ 4 & \infty & 0 & 8 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

Reducem linia 3 cu 3

$$\begin{pmatrix} \infty & 0 & 5 & 0 \\ 4 & \infty & 0 & 8 \\ 1 & 5 & \infty & 0 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

Reducem linia 4 cu 2

$$\begin{pmatrix} \infty & 0 & 5 & 0 \\ 4 & \infty & 0 & 8 \\ 1 & 5 & \infty & 0 \\ 4 & 0 & 4 & \infty \end{pmatrix}$$

Reducem coloana 1 cu 1

$$\begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

Circuit hamiltonian minim TSP

$$C = \begin{pmatrix} \infty & 3 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

matricea redusă = $\begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$

Cantitatea totală cu care s-a redus matricea =

2 + 1 + 3 + 2 + 1 = 9

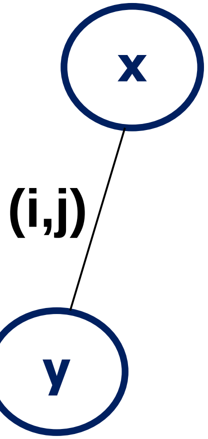
Algoritmul Branch and Bound pentru TSP

- f(rad) = cantitatea totală cu care s-a redus matricea C

- Unui vârf x îi asociem
  - o matrice de costuri redusă M_x (dacă nu este frunză)
  - o valoare f(x) calculată după cum urmează

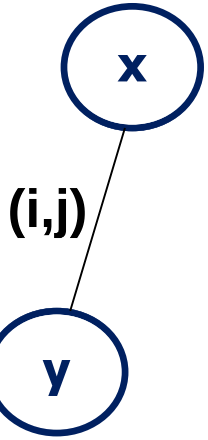
Algoritmul Branch and Bound pentru TSP

- Pentru un vârf y din arborele BB al cărui tată este x şi muchia (x,y) este etichetată cu (i,j) modificăm M_x:



Algoritmul Branch and Bound pentru TSP

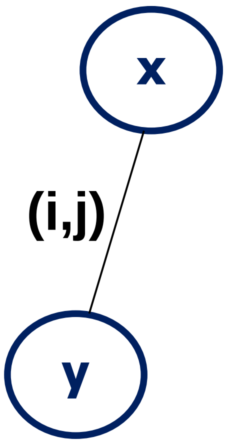
- Pentru un vârf y din arborele BB al cărui tată este x şi muchia (x,y) este etichetată cu (i,j) modificăm M_x:



  - elementele liniei i devin ∞ (mergem sigur către j)
  - elementele coloanei j devin ∞ (am ajuns sigur în j)
  - $M_x(j,1) \leftarrow \infty$ pentru a nu reveni prematur în rădăcina 1;

Algoritmul Branch and Bound pentru TSP

- Pentru un vârf y din arborele BB al cărui tată este x şi muchia (x,y) este etichetată cu (i,j) modificăm M_x:



  - elementele liniei i devin ∞ (mergem sigur către j)
  - elementele coloanei j devin ∞ (am ajuns sigur în j)
  - $M_x(j,1) \leftarrow \infty$ pentru a nu reveni prematur în rădăcina 1;
  - reducem noua matrice M_x şi obţinem M_y;

    fie r cantitatea cu care s-a redus M_x.
  - Luăm $\mathbf{f(y) \leftarrow f(x) + M_x(i,j) + r}$.

Algoritmul Branch and Bound pentru TSP

- Avem

  $\mathbf{f(x)} \leq$ cost CHM corespunzător lui x

- Dacă x este frunză avem

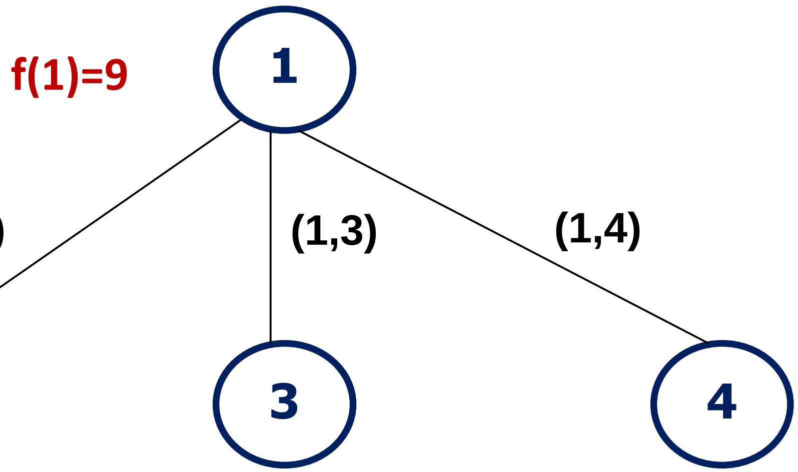
  f(x) = c(x) = cost CHM

$$C = \begin{pmatrix} \infty & 2 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

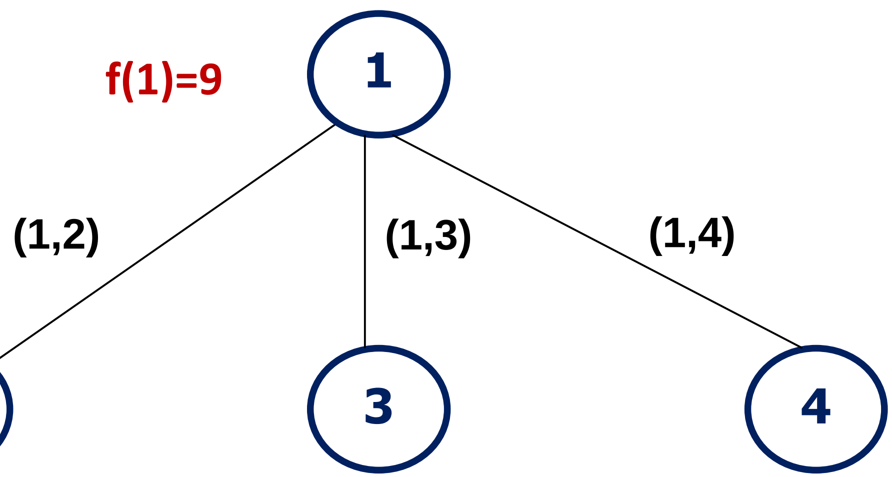
$$M_1 = \begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

$$f(1) = 2 + 1 + 3 + 2 + 1 = 9$$


f(1)=9
1



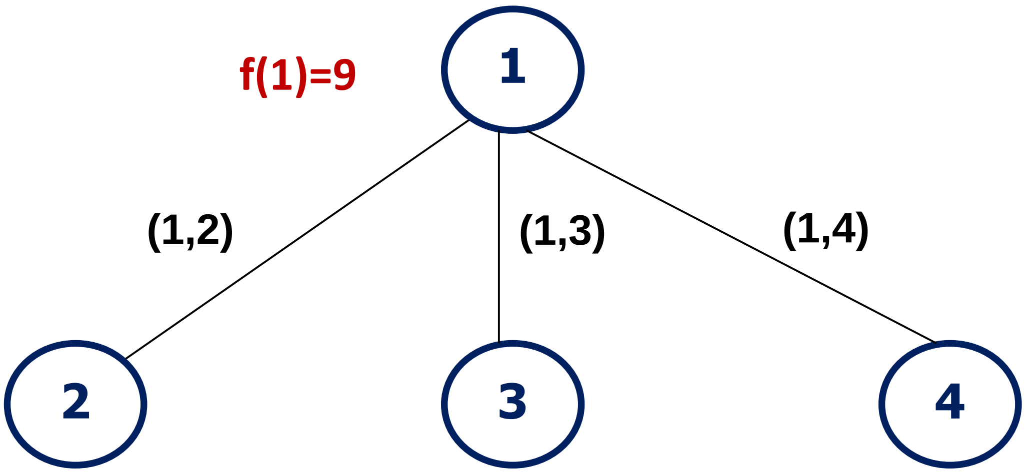
L = {2, 3, 4}



L = {2, 3, 4}

Calculăm f(2)

$$M_1 = \begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

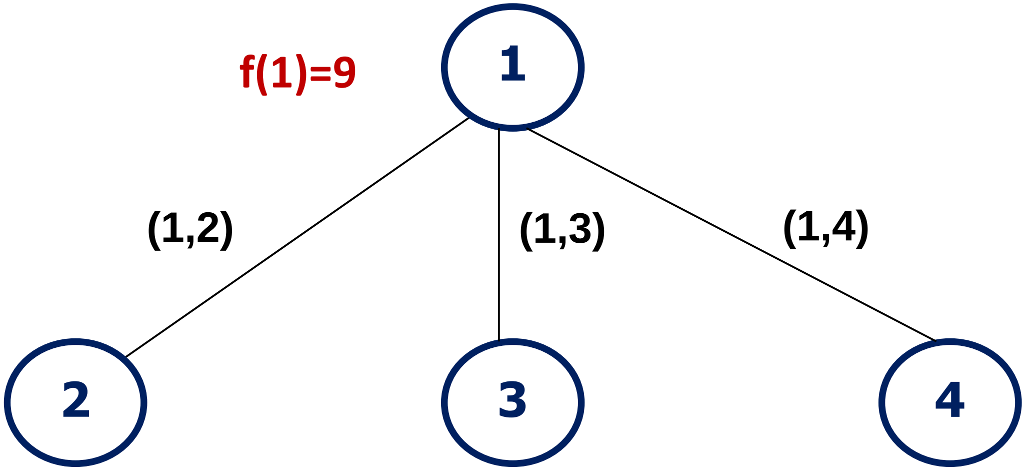


L = {2, 3, 4}

Calculăm f(2)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 \\ 0 & \infty & \infty & 0 \\ 3 & \infty & 4 & \infty \end{pmatrix}$$

- Linia 1 și coloana 2 devin ∞
- Elementul (2, 1) devine ∞
- Reducem linia 4 cu 3

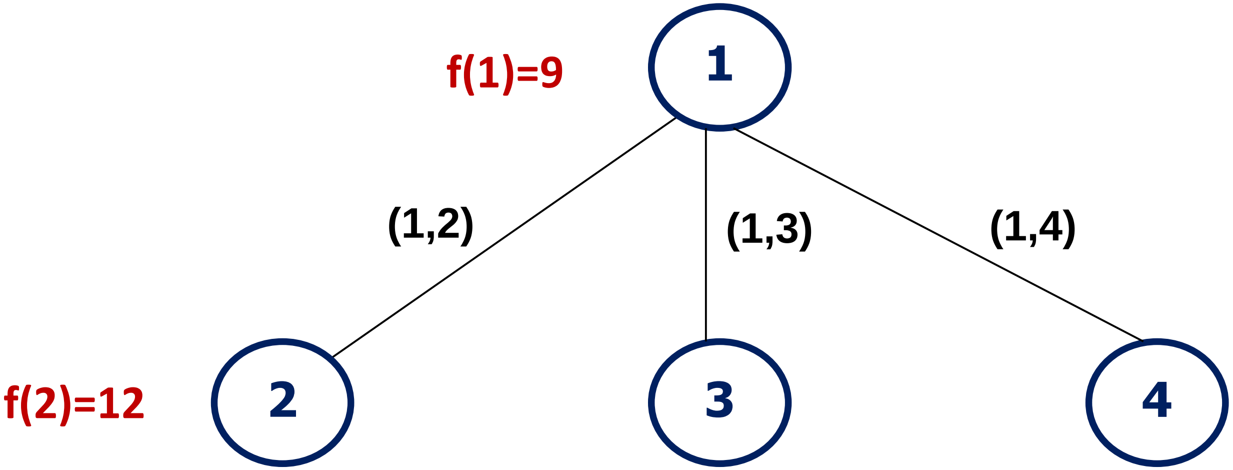


L = {2, 3, 4}

Calculăm f(2)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 \\ 0 & \infty & \infty & 0 \\ 0 & \infty & 1 & \infty \end{pmatrix}$$

- Linia 1 și coloana 2 devin ∞
- Elementul (2, 1) devine ∞
- Reducem linia 4 cu 3

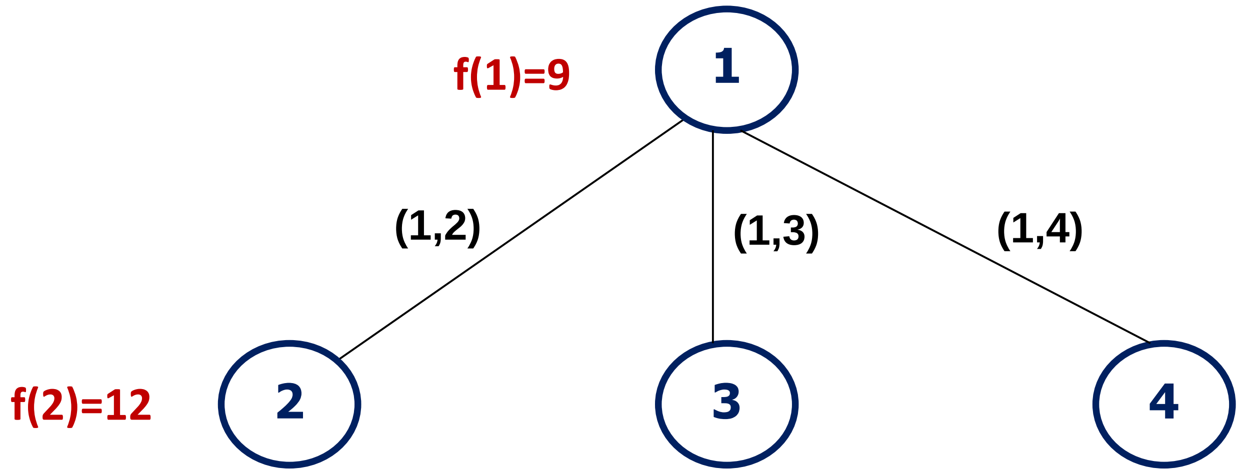


L = {2, 3, 4}

Calculăm f(2)

$$M_2 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 8 \\ 0 & \infty & \infty & 0 \\ 0 & \infty & 1 & \infty \end{pmatrix}$$

- Linia 1 și coloana 2 devin ∞
- Elementul (2, 1) devine ∞
- Reducem linia 4 cu 3
- Obținem $f(2) = f(1)+r + M_1(1,2) = 9+3+0=12$

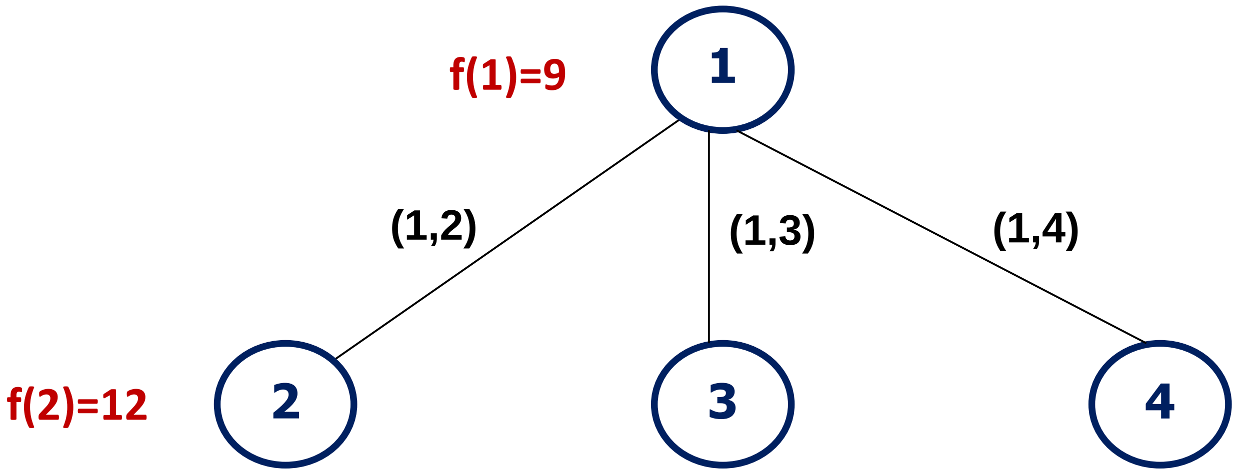


L = {2, 3, 4}

Calculăm f(3)

$$M_1 = \begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

- Linia 1 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞

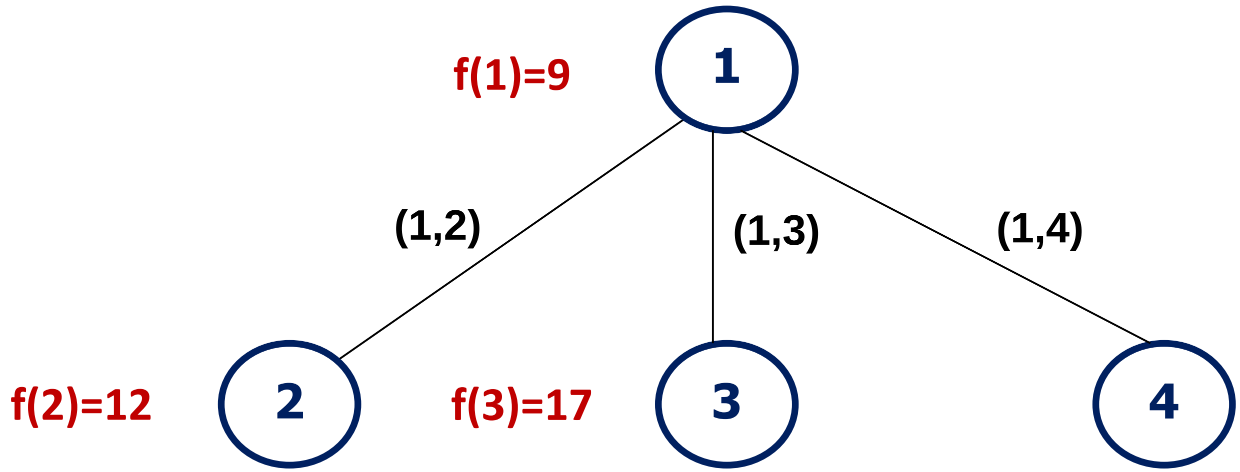


L = {2, 3, 4}

Calculăm f(3)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & 8 \\ \infty & 5 & \infty & 0 \\ 3 & 0 & \infty & \infty \end{pmatrix}$$

- Linia 1 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞
- Reducem linia 2 cu 3

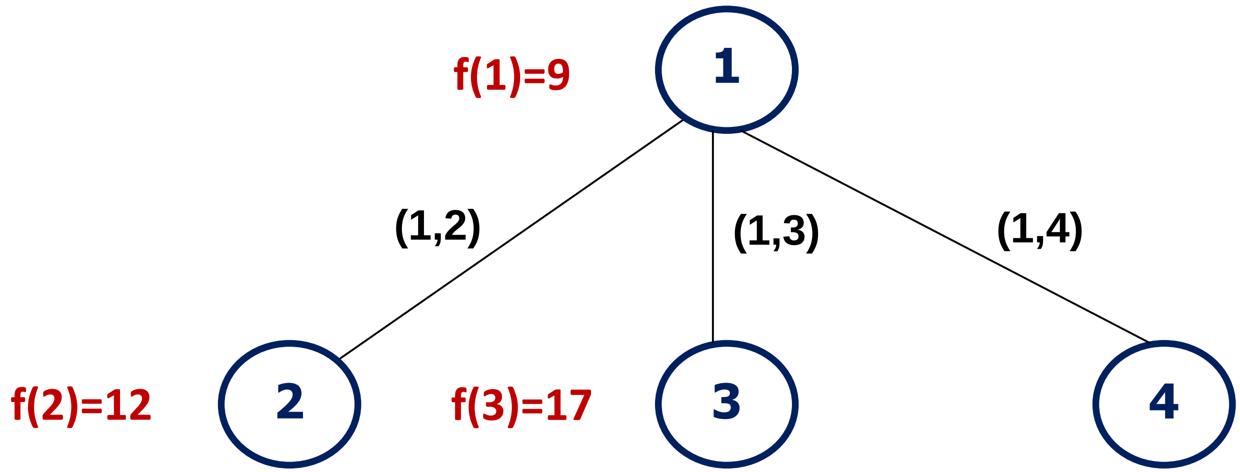


L = {2, 3, 4}

Calculăm f(3)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 5 \\ \infty & 5 & \infty & 0 \\ 3 & 0 & \infty & \infty \end{pmatrix}$$

- Linia 1 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞
- Reducem linia 2 cu 3
- Obținem $f(3) = f(1)+r + M_1(1,3) = 9+3+5=17$

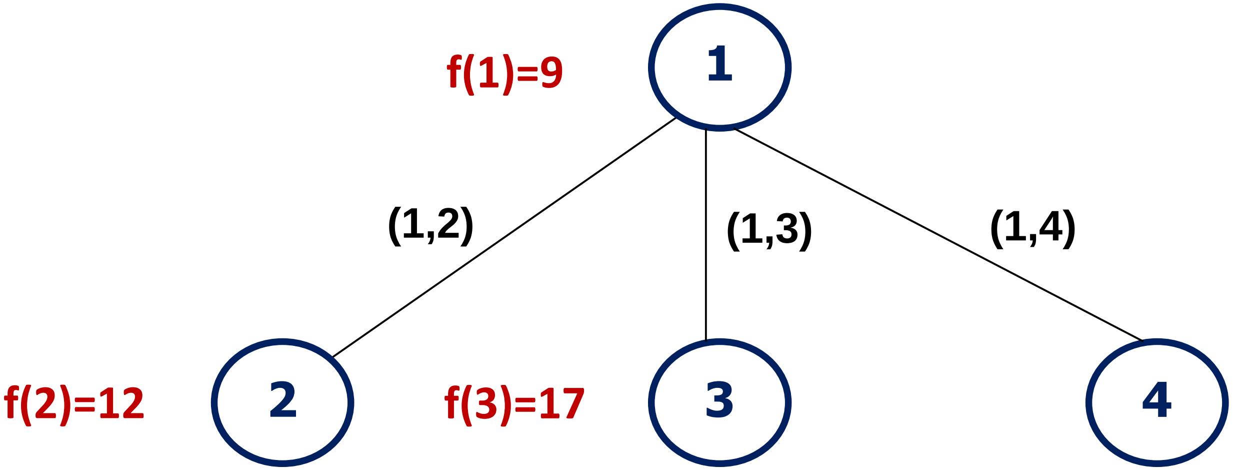


L = {2, 3, 4}

Calculăm f(4)

$$M_1 = \begin{pmatrix} \infty & 0 & 5 & 0 \\ 3 & \infty & 0 & 8 \\ 0 & 5 & \infty & 0 \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

- Linia 1 și coloana 4 devin ∞
- Elementul (4, 1) devine ∞

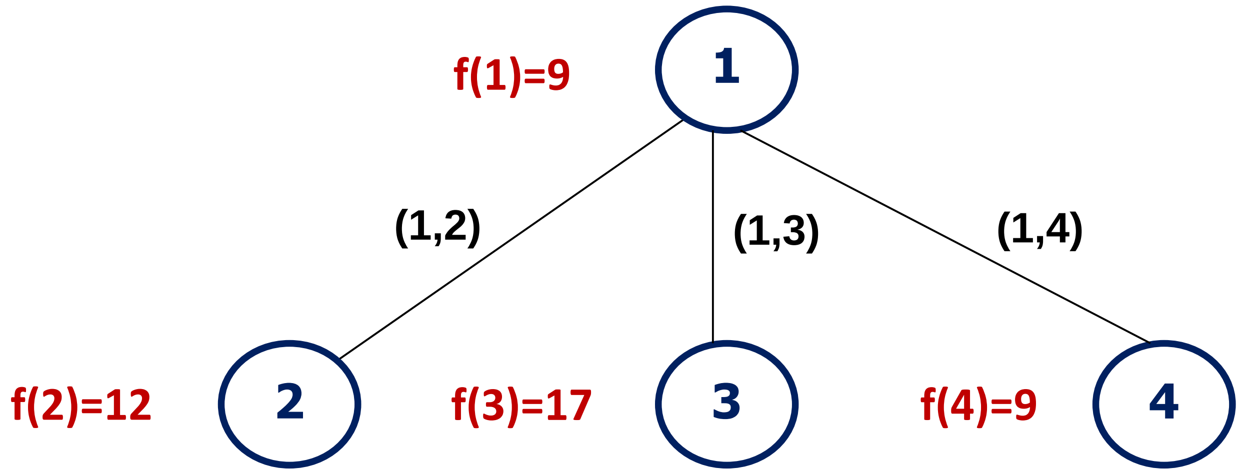


L = {2, 3, 4}

Calculăm f(4)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & 0 & \infty \\ 0 & 5 & \infty & \infty \\ \infty & 0 & 4 & \infty \end{pmatrix}$$

- Linia 1 și coloana 4 devin ∞
- Elementul (4, 1) devine ∞

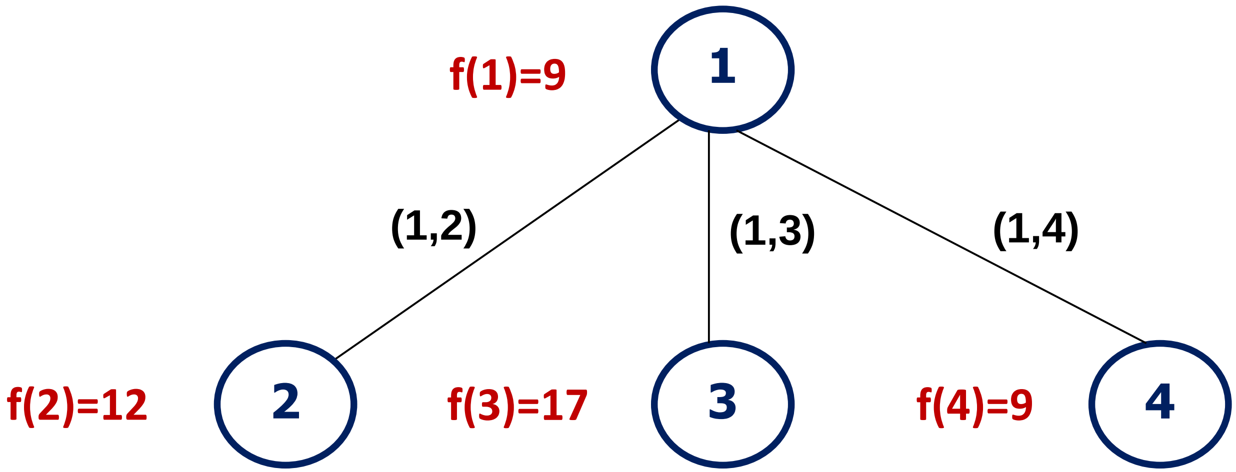


L = {2, 3, 4}

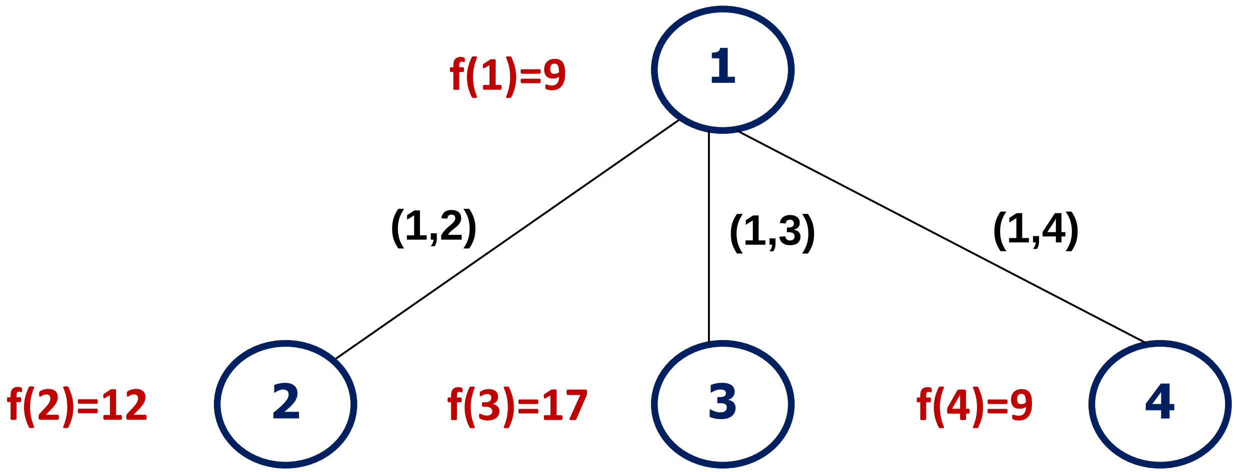
Calculăm f(4)

$$M_4 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & 0 & \infty \\ 0 & 5 & \infty & \infty \\ \infty & 0 & 4 & \infty \end{pmatrix}$$

- Linia 1 și coloana 4 devin ∞
- Elementul (4, 1) devine ∞
- Nu sunt necesare reduceri
- Obținem $f(4) = f(1) + M_1(1,4) = 9 + 0 = 9$

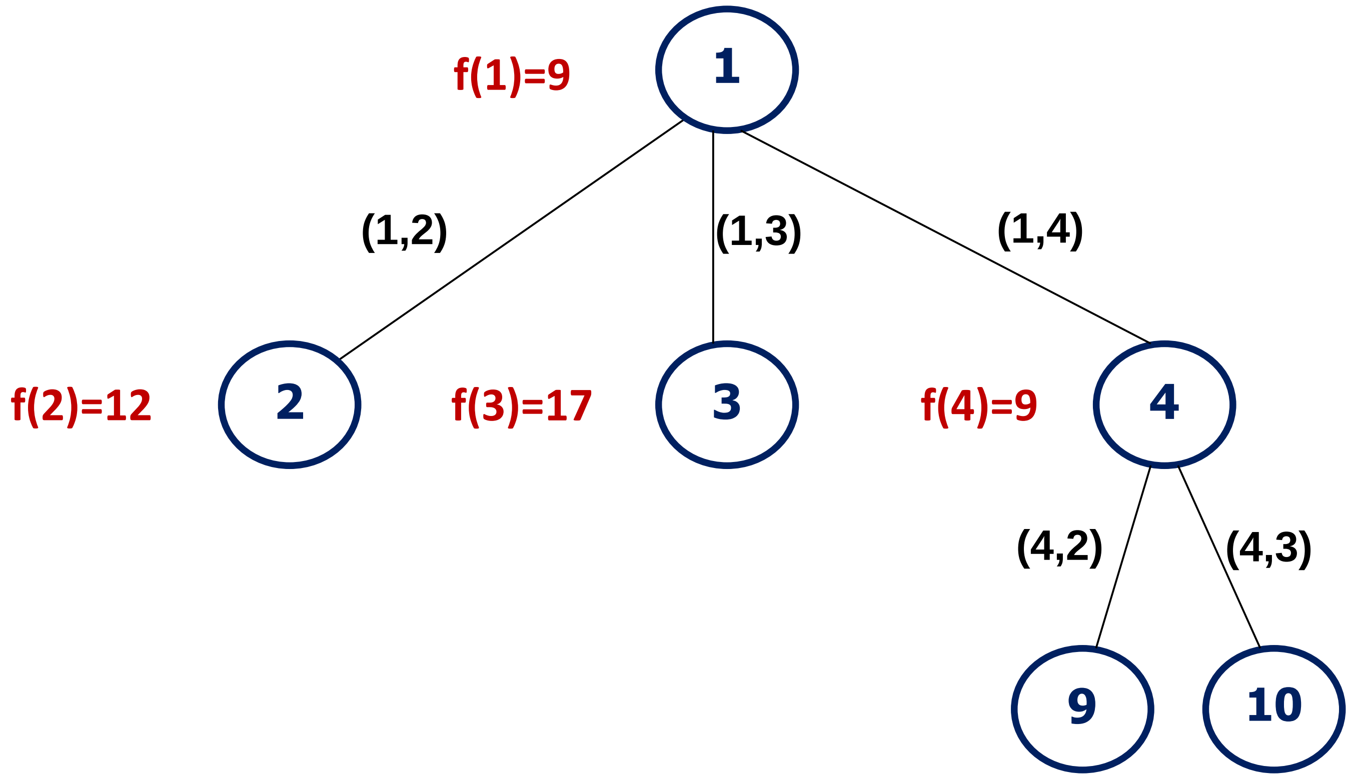


L = {2, 3, 4}



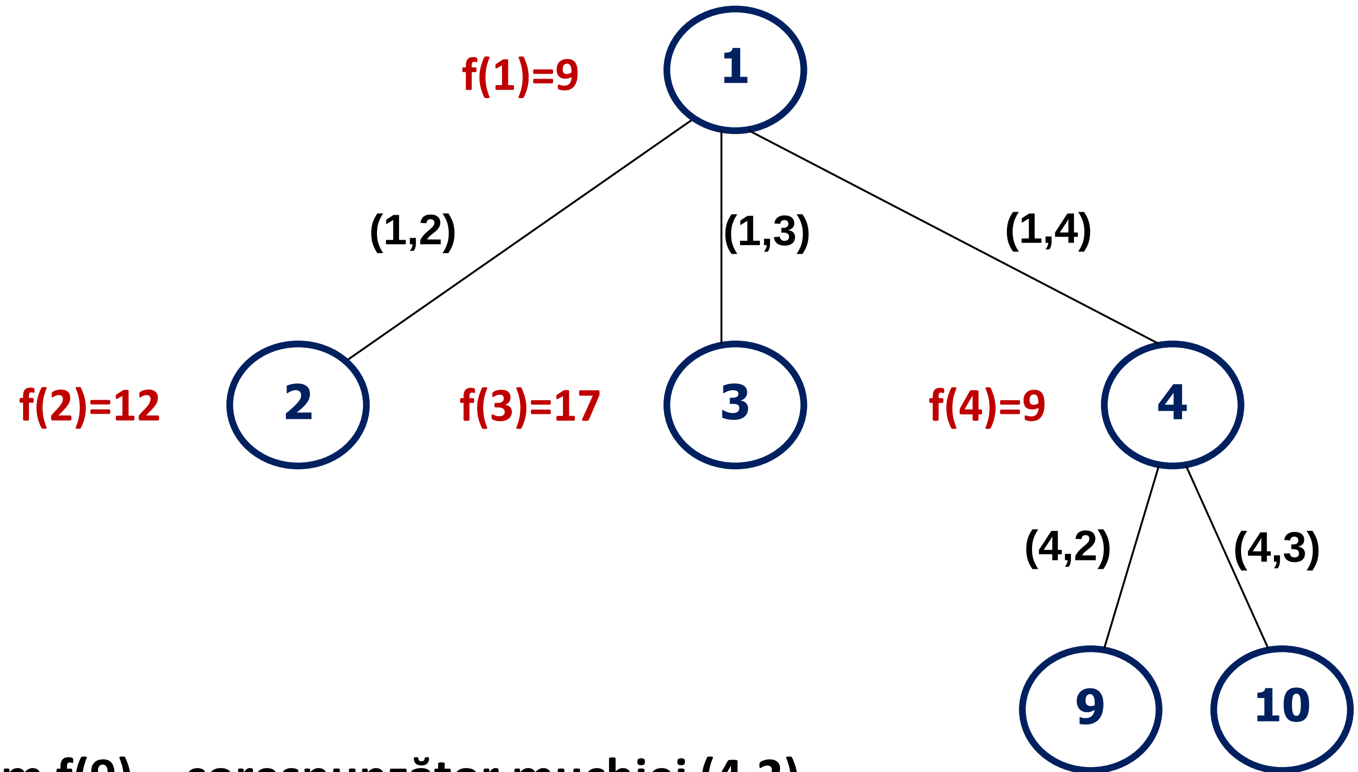
L = {2, 3, 4}

Extragem din L vârful cu f minim -> 4



L = {2, 3, 9, 10}

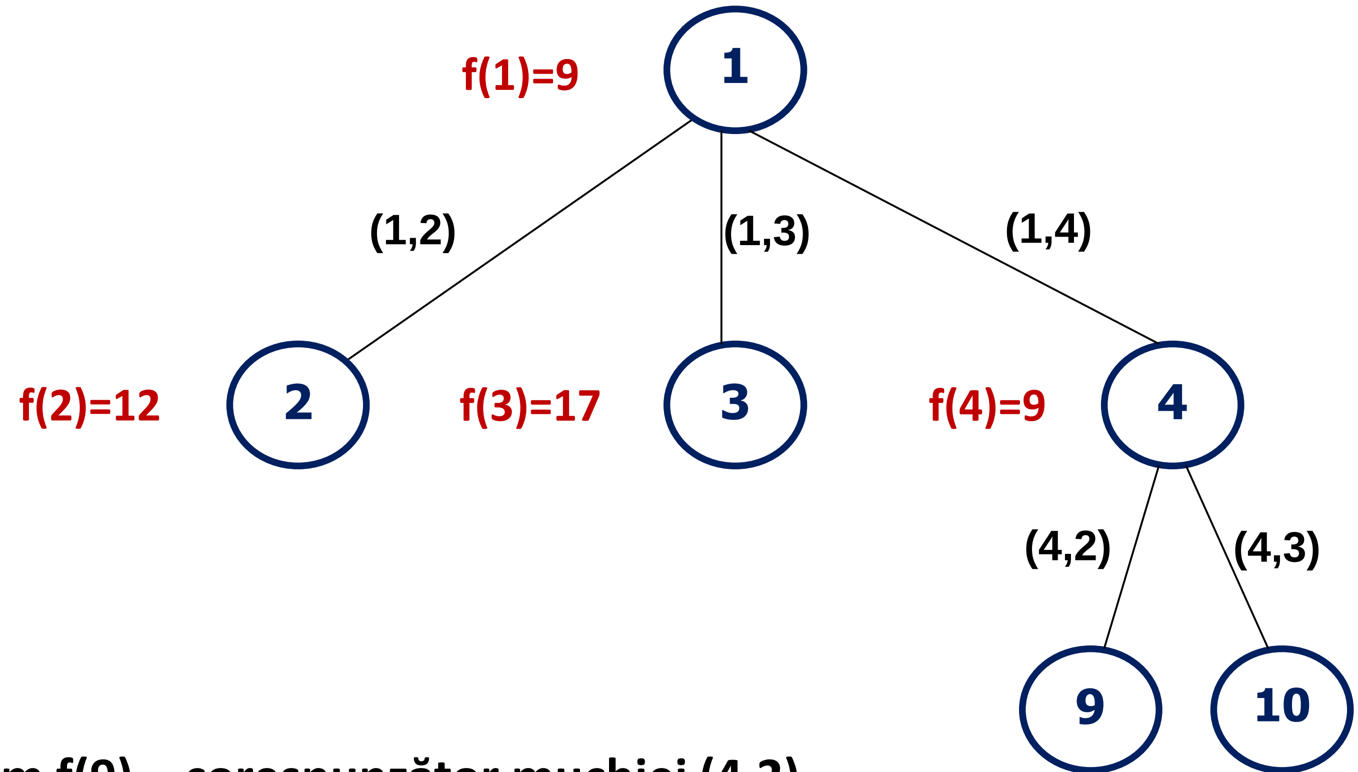
Calculăm f(9) și f(10) analog (pornind de la M_4)



Calculăm f(9) – corespunzător muchiei (4,2)

$$M_4 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & 0 & \infty \\ 0 & 5 & \infty & \infty \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

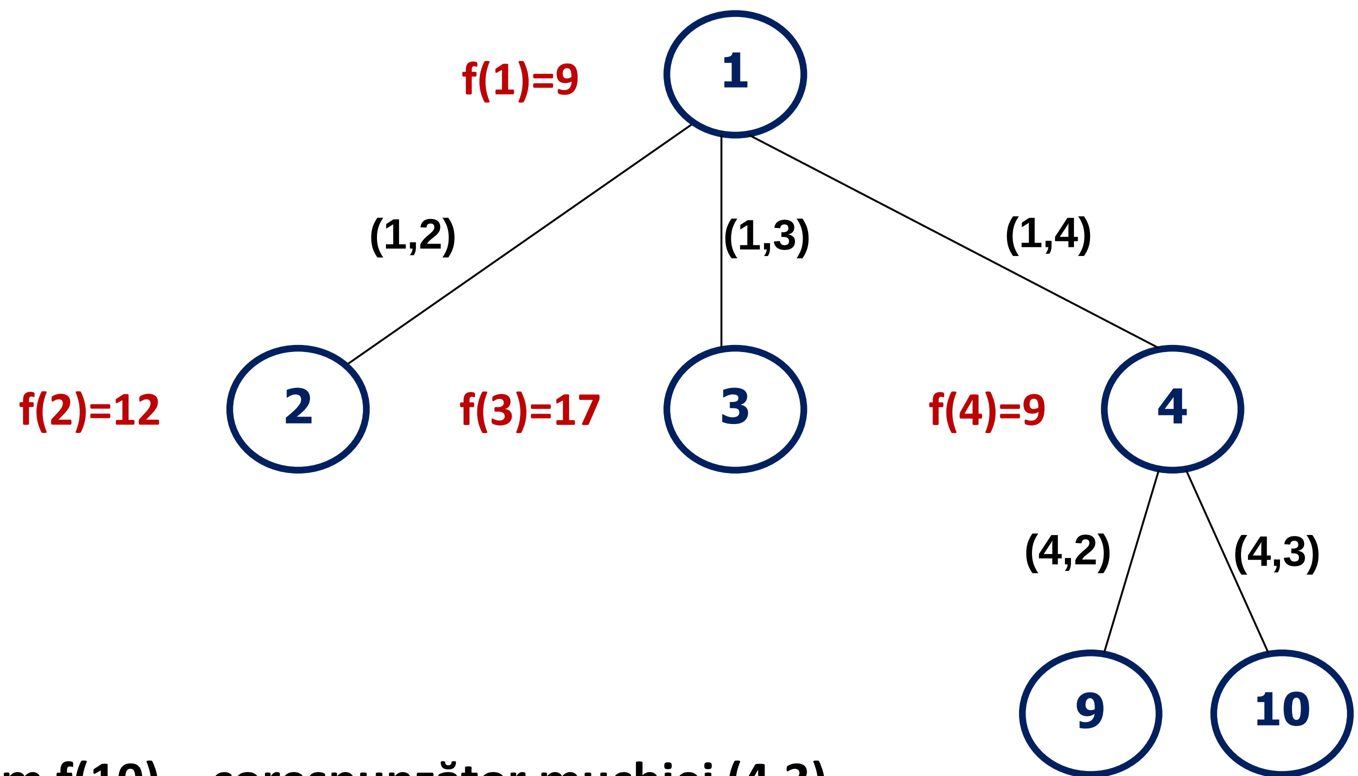
- Linia 4 și coloana 2 devin ∞
- Elementul (2, 1) devine ∞



Calculăm f(9) – corespunzător muchiei (4,2)

$$M_9 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

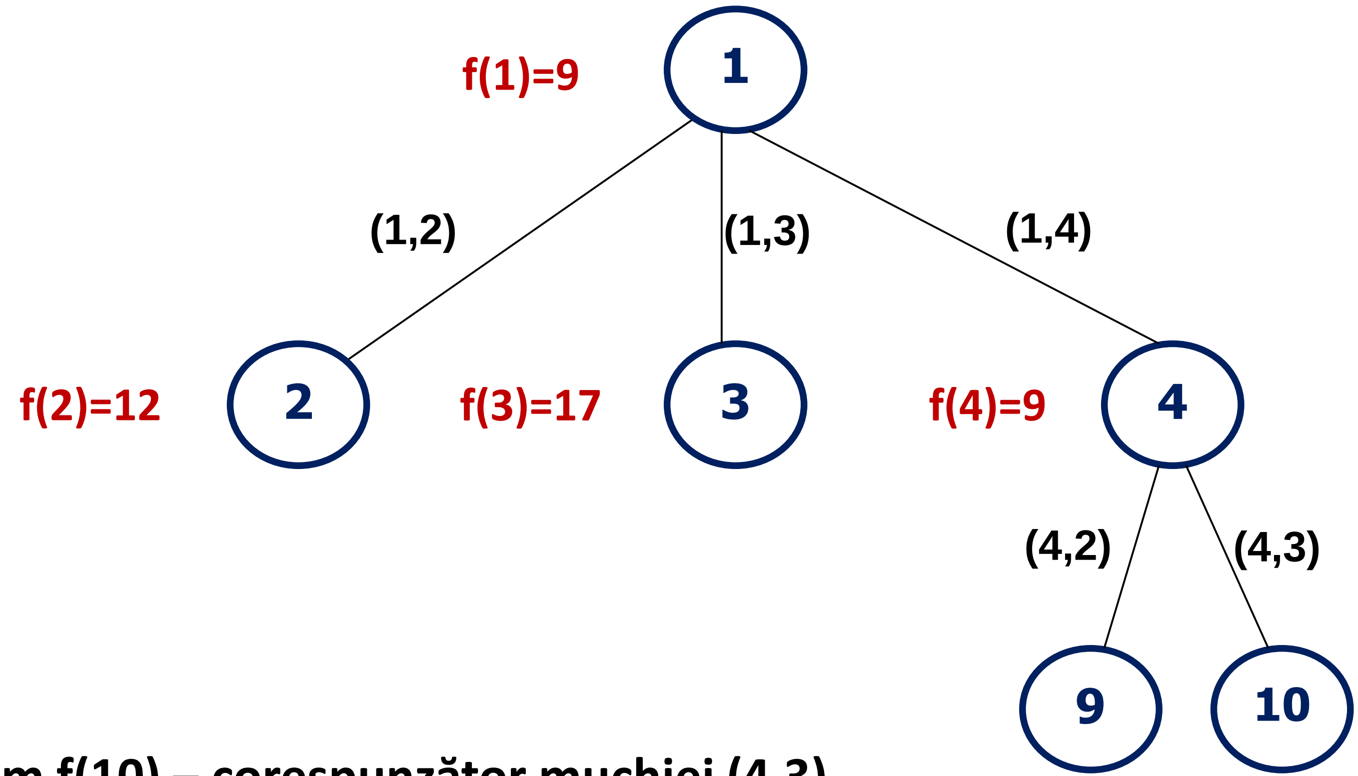
- Linia 4 și coloana 2 devin ∞
- Elementul (2, 1) devine ∞
- Nu sunt necesare reduceri
- $f(9) = f(4) + M_4(4,2) = 9 + 0 = 9$



Calculăm f(10) – corespunzător muchiei (4,3)

$$M_4 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & 0 & \infty \\ 0 & 5 & \infty & \infty \\ 3 & 0 & 4 & \infty \end{pmatrix}$$

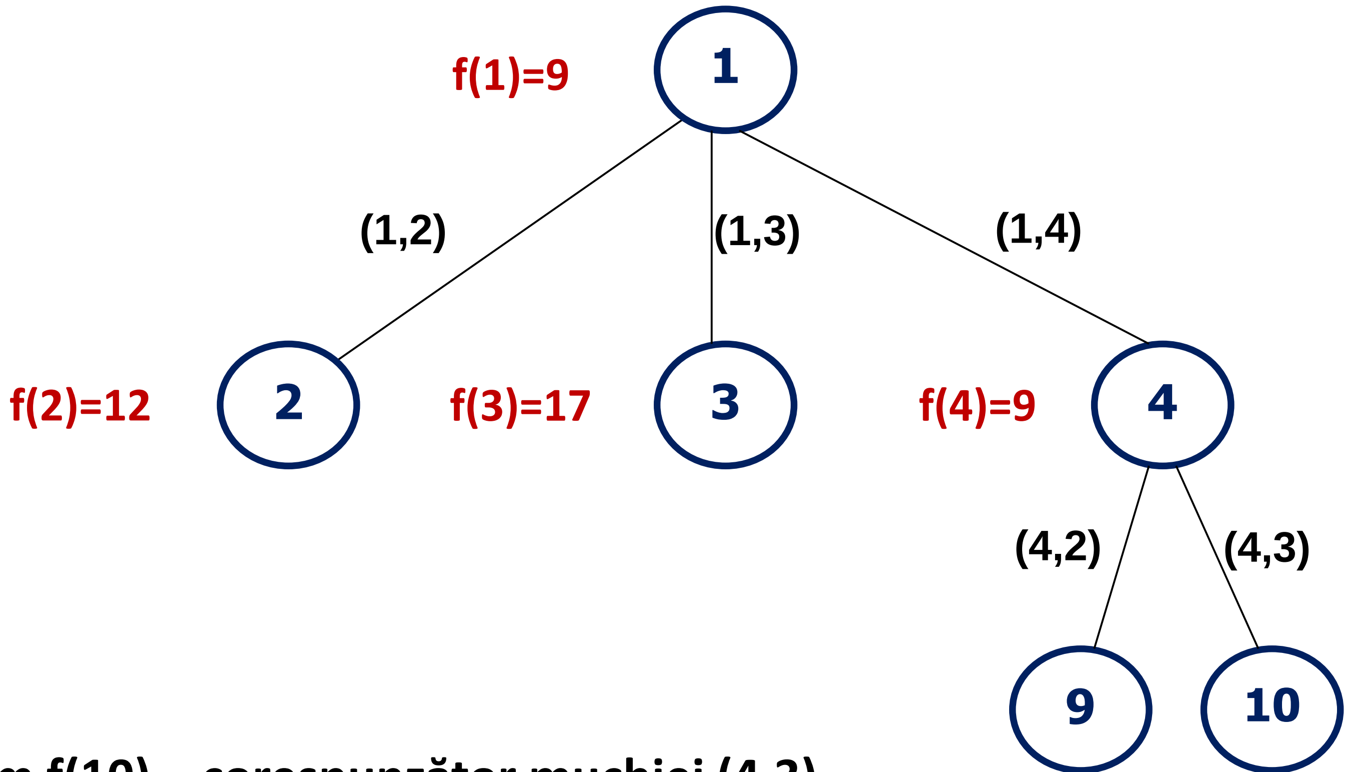
- Linia 4 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞



Calculăm f(10) – corespunzător muchiei (4,3)

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty \\ \infty & 5 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

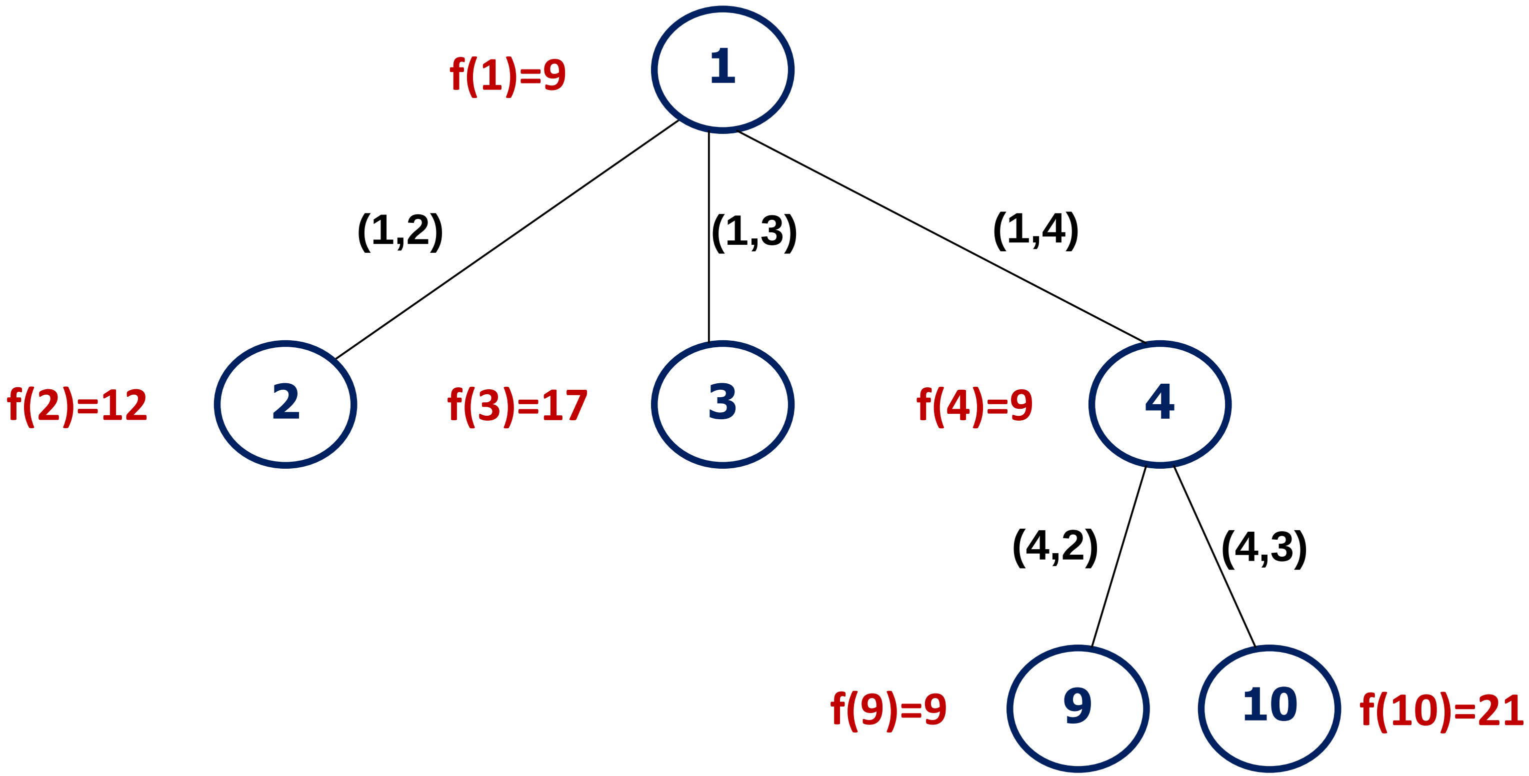
- Linia 4 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞
- Reducem linia 2 cu 3 și linia 3 cu 5



Calculăm f(10) – corespunzător muchiei (4,3)

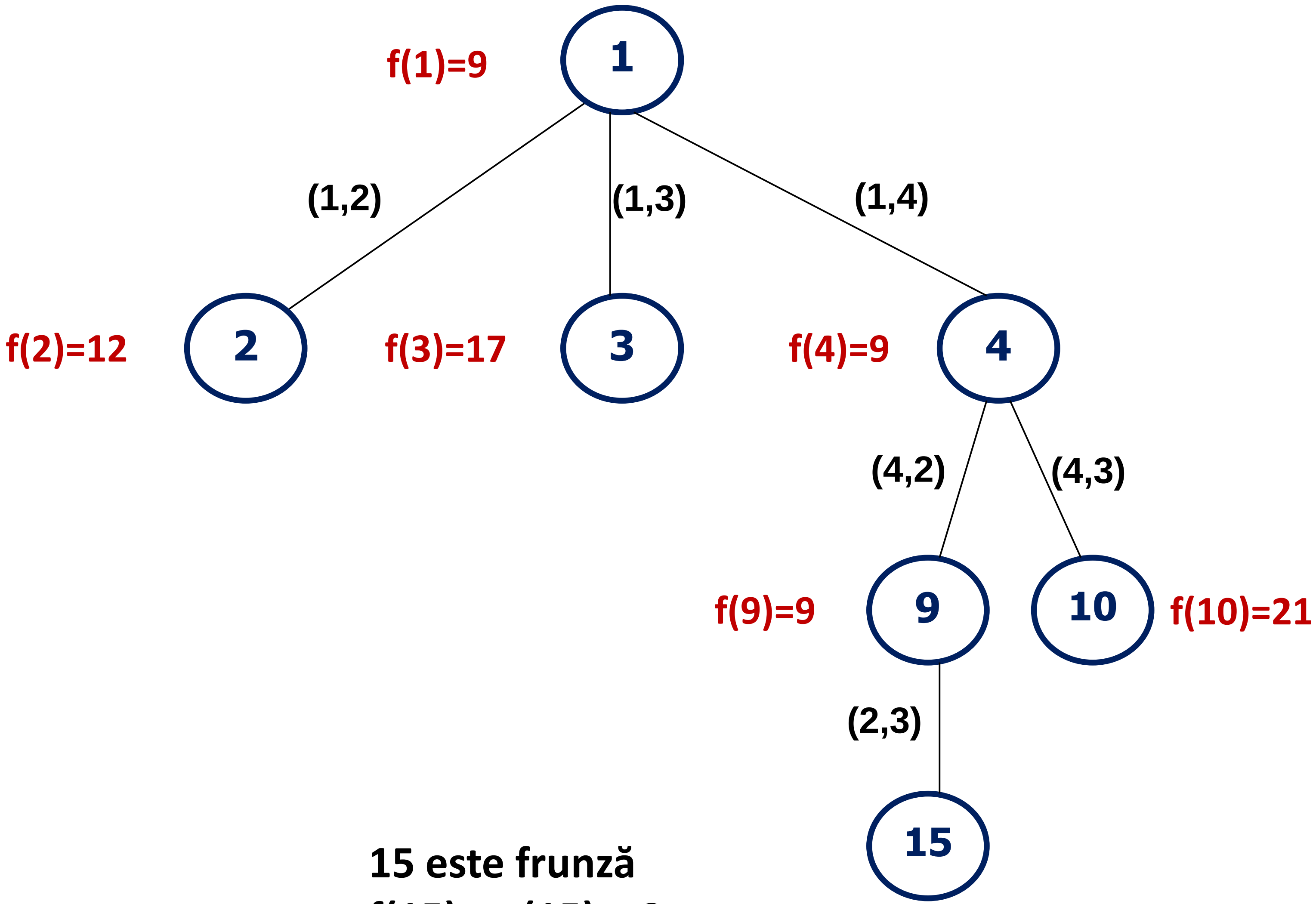
$$M_{10} = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

- Linia 4 și coloana 3 devin ∞
- Elementul (3, 1) devine ∞
- Reducem linia 2 cu 3 și linia 3 cu 5
- $f(10)=f(4)+r+M_4(4,3)=9+(3+5)+4=21$



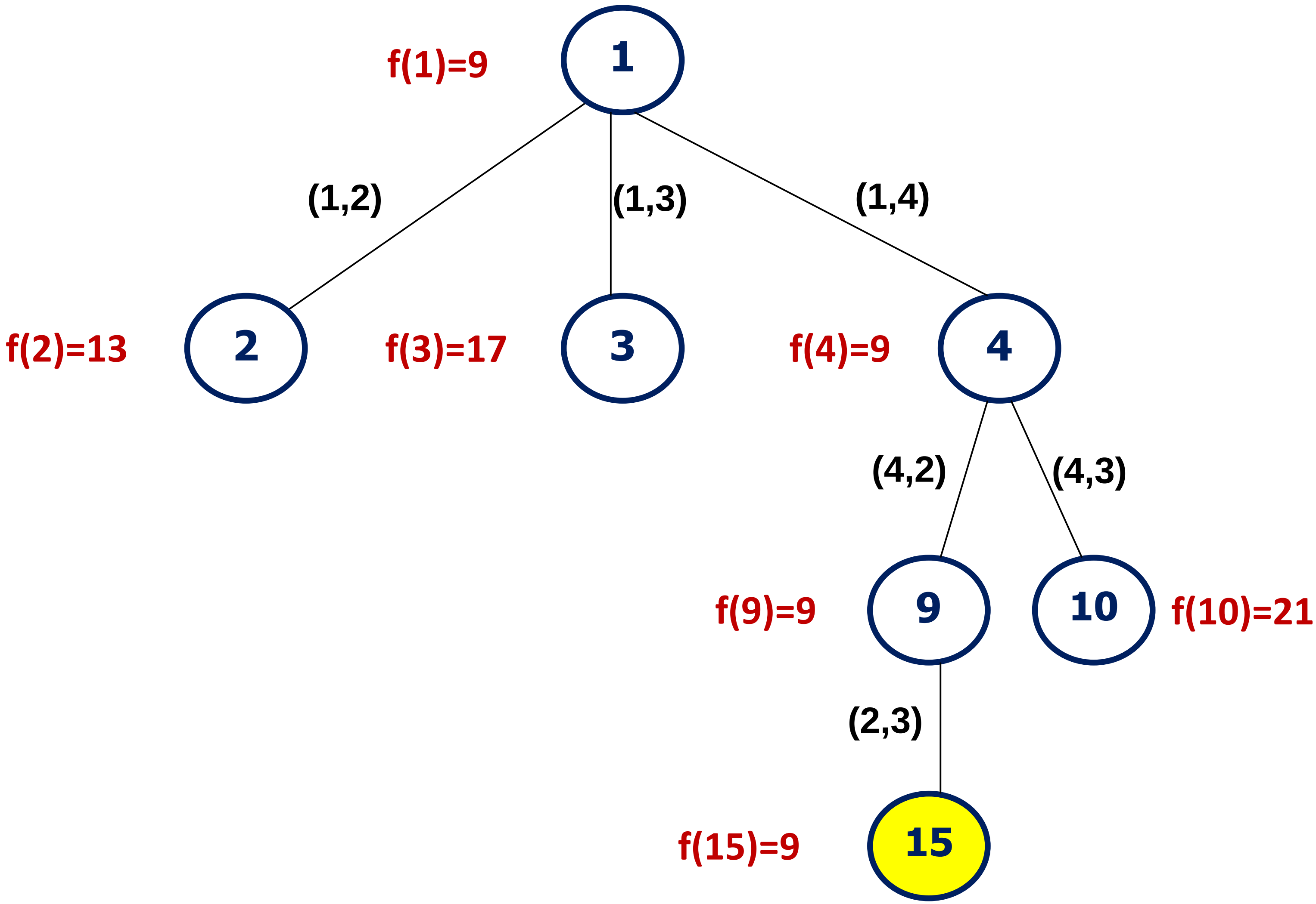
L = {2, 3, 9, 10}

Extragem din L vârful cu f minim -> 9



15 este frunză

f(15) = c(15) = 9



- min devine 9
- se elimină din L toate vârfurile cu f mai mare decât 9
- L devine vidă -> STOP