Capitolul 9

Tehnici de parcurgere a spaţiului soluţiilor

Multe dintre problemele întâlnite în informatică se bazează pe căutarea în spaţiul soluţiilor. In această categorie intră atât problemele care necesită identificarea unei configuraţii care satisface anumite restricţii cât şi cele care urmăresc optimizarea unui criteriu. Deşi spaţiul soluţiilor este finit, dimensiunea acestuia creşte de regulă exponenţial cu dimensiunea problemei (cum se întâmplă de exemplu în cazul în care spaţiul soluţiilor potenţiale ale unei probleme de dimensiune n este reprezentat de ansamblul tuturor submulţimilor unei mulţimi cu n elemente). Generarea tuturor configuraţiilor care sunt potenţiale soluţii şi alegerea dintre acestea a celor ce satisfac restricţiile şi/sau criteriul de optim este o abordare total ineficientă.

În astfel de situaţii este necesară o tehnică de căutare controlată a spaţiului soluţiilor, care sa evite pe cât posibil redundanţă şi să permite abandonarea unor configuraţii în momentul în care se poate decide că acestea nu conduc la soluţii ale problemei. Astfel de tehnici sunt *căutarea cu revenire* (backtracking) şi cea de tip *ramifică şi mărgineşte* (branch and bound). Principiile celor două tehnici sunt asemănătoare însă diferă domeniile de aplicabilitate. Căutarea cu revenire este folosită în special pentru probleme de satisfacere a restricţiilor pe când căutarea de tip "ramifică şi mărgineşte" este folosită pentru rezolvarea problemelor de optimizare.

Trebuie menţionat faptul că deşi aceste tehnici conduc de regulă la algoritmi mai eficienţi decât tehnica forţei brute, totuşi complexitatea lor este ridicată astfel că pot fi aplicaţi efectiv doar pentru probleme de dimensiune relativ mică. Specific acestor tehnici este faptul că pentru anumite instanţe ale unei probleme este posibil ca soluţia problemei să fie obţinută prin parcurgerea unei porţiuni mici a spaţiului soluţiilor pe când pentru alte instanţe ale aceleiaşi probleme este posibil ca porţiunea parcursă să fie semnificativ mai mare.

9.1 Principiul căutării cu revenire

Tehnica căutării cu revenire se utilizează pentru rezolvarea problemelor printr-o parcurgere controlată a spaţiului soluţiilor. În ultimă instanţă este o îmbunătăţire a metodei căutării exhaustive (metoda forţei brute) care permite reducerea numărului de soluţii potenţiale analizate.

Majoritatea problemelor ce pot fi rezolvate prin "backtracking" pot fi reduse la determinarea unei submulţimi a unui produs cartezian de forma $A_1 \times A_2 \times \ldots \times A_n$ (cu mulţimile A_k finite). Fiecare element al submulţimii poate fi văzut ca o soluţie (metoda fiind astfel adecvată în special în situaţiile în care se doreşte determinarea tuturor soluţiilor unei probleme, nu numai a uneia dintre ele). O soluţie este de forma $s = (s_1, s_2, \ldots, s_n)$ cu $s_k \in A_k = \{a_1^k, a_2^k, \ldots, a_{m_k}^k\}$, $m_k = \text{card}A_k$. În majoritatea cazurilor nu orice element al produsului cartezian este soluţie ci doar cele care satisfac anumite *restricţii*. De exemplu, problema determinării tuturor permutărilor de ordin n poate fi reformulată ca problema determinării submulţimii produsului cartezian $\{1, 2, \ldots, n\} \times \ldots \{1, 2, \ldots, n\}$ ($A_1 = A_2 = \ldots = A_n = \{1, 2, \ldots, n\}$) în care elementele au componentele distincte (restricţiile problemei sunt: $s_i \neq s_j$, pentru orice $i \neq j$).

O soluţie s se obţine completând succesiv componentele s_k pentru $k = \overline{1, n}$. Specificul metodei constă în maniera de parcurgere a spaţiului soluţiilor:

- Soluţiile sunt construite succesiv, la fiecare etapă fiind completată câte o componentă (similar cu tehnica greedy însă ulterior se poate reveni asupra alegerii unei componente);

- Alegerea unei valori pentru o componentă se face într-o anumită ordine (aceasta presupune că pe mulţimile A_k există o relaţie de ordine şi se realizează o parcurgere sistematică a spaţiului $A_1 \times A_2 \times \ldots \times A_n$);

- La completarea componentei k se verifică dacă soluţia parţială (s_1, s_2, ..., s_k), verifică condiţiile induse de restricţiile problemei (acestea sunt numite *condiţii de continuare*). O soluţie parţială care satisface condiţiile de continuare este denumită soluţie parţială validă (sau viabilă) întrucât poate conduce la o soluţie a problemei.

- Dacă au fost încercate toate valorile corespunzătoare componentei k şi încă nu a fost găsită o soluţie sau dacă se doreşte determinarea unei noi soluţii atunci se revine la componenta anterioară $(k - 1)$ şi se încearcă următoarea valoare corespunzătoare acesteia ş.a.m.d. Această revenire la o componentă anterioară este specifică tehnicii backtracking.

- Procesul de căutare şi revenire este continuat fie până când este găsită o soluţie (în cazul în care este suficientă determinarea uneia) sau până când au fost testate toate configuraţiile posibile.

Strategia de construire a soluţiilor aplicând backtracking este similară construirii unui structuri arborescente ale cărei noduri corespund unor soluţii parţiale valide (nodurile interne) sau soluţiilor finale (nodurile de pe frontieră). Nodurile de pe frontieră pot corespunde unor soluţii parţiale invalide. În figura 9.1 este ilustrat modul de parcurgere a spaţiului soluţiilor în cazul generării permutărilor de ordin 3. Ramurile abandonate în momentul în care condiţiile de continuare nu sunt satisfăcute sunt marcate cu `X`.

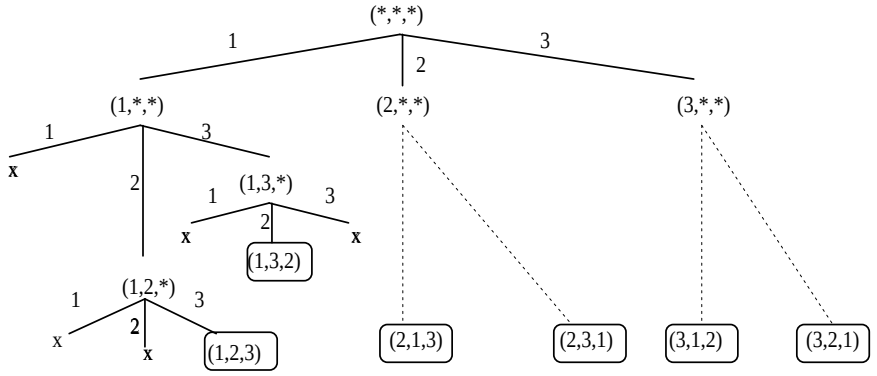

Figura 9.1: Ilustrarea parcurgerii spaţiului soluţiilor în cazul generării permutărilor

În cazul în care sunt specificate restricţii, anumite ramuri ale structurii arborescente asociate parcurgerii sunt abandonate înainte de a atinge lungimea maximă. În aplicarea metodei pentru o problemă concretă se parcurg etapele:

- se alege o reprezentare a soluţiei sub forma unui vector cu n componente;
- se identifică mulţimile A_1, A_2, ..., A_n şi relaţiile de ordine care indică modul de parcurgere a fiecărei mulţimi;
- pornind de la restricţiile problemei se stabilesc condiţiile de validitate ale soluţiilor parţiale (condiţiile de continuare).
- se stabileşte criteriul în baza căruia se poate decide că s-a obţinut o soluţie finală.

În descrierea structurii generale a algoritmului vom folosi notaţiile: $A_k = \{a_1^k, \ldots, a_{m_k}^k\}$, m_k reprezentând numărul de elemente din A_k; k indică componenta curentă din s; i_k reprezintă indicele elementului curent din A_k. Structura generală a algoritmului este descrisă în 9.1.
Observaţii. În aplicaţiile concrete pot interveni următoarele situaţii:

Algoritmul 9.1 Structura generală a unui algoritm bazat pe tehnica căutării cu revenire

```
backtracking(n, A_1,..., A_n)
k ← 1          // se începe cu completarea primei componente
i_k ← 0        // se pregăteşte indicele de parcurgere a lui A_k
while k > 0 do
  i_k ← i_k + 1          // indicele următorului element din A_k
  valid ← false
          // căutarea unei componente valide pentru poziţia k
  while (valid =false) and (i_k ≤ m_k) do
    s_k ← a^k_{i_k}          // se încearcă valoarea curentă din A_k
    if (s_1, ..., s_k) "satisface condiţiile de continuare" then
      valid ← true
    else
      i_k ← i_k + 1
    end if
  end while
  if valid =true then
    if (s_1, ..., s_k) este soluţie then
      "s-a gasit o soluţie"
    else
      k ← k + 1;
      i_k ← 0 // se pregăteşte completarea următoarei componente
    end if
  else
    k ← k − 1 // se revine la completarea componentei anterioare
  end if
end while
```

1. Mulţimile $A_1,\ldots, A_n$ nu sunt neapărat distincte sau elementele lor pot fi generate pe parcursul algoritmului fără a fi necesară transmiterea lor ca argument algoritmului.

2. Condiţiile de continuare se deduc din restricţiile problemei.

3. Pentru unele probleme soluţia este obţinută în cazul în care $k = n$ iar pentru altele este posibil să fie satisfăcută o condiţie de găsire a soluţiei pentru $k < n$ (pentru anumite probleme nu toate soluţiile conţin acelaşi număr de elemente).

4. La găsirea unei soluţii de cele mai multe ori aceasta se afişează sau se reţine într-o zonă dedicată. Dacă se doresţe identificarea unei singure soluţii atunci căutarea se opreşte după găsirea acesteia, altfel procesul de căutare continuă prin analizarea următoarei valori a ultimei componente

completate.

Exemplul 9.1 *Generarea permutărilor de ordin n.* Caracteristicile acestei probleme sunt:

Reprezentarea soluţiilor. O permutare de ordin n este un tablou cu n elemente *distincte* din $\{1, 2, \ldots, n\}$.

Mulţimile A_k. Mulţimile ce conţin componentele soluţiilor sunt toate egale cu $\{1, 2, \ldots, n\} = A_1 = \ldots = A_n$. Ordinea de parcurgere a elementelor este cea naturală: de la cel mai mic către cel mai mare element.

Restricţiile şi condiţiile de continuare. Dacă $s = (s_1, \ldots, s_n)$ este o soluţie atunci ea trebuie să respecte restricţiile: $s_i \neq s_j$ pentru oricare $i \neq j$. Un vector cu k elemente, $(s_1, s_2, \ldots, s_k)$, poate conduce la o soluţie doar dacă satisface condiţiile de continuare $s_i \neq s_j$ pentru orice $i \neq j$ $(i, j \in \{1, \ldots, k\})$. Vom considera că verificarea condiţiilor de continuare va fi efectuată în cadrul unui algoritm de validare. Se observă că la completarea componentei k este suficient să se verifice că s_k este diferit de s_1, s_2, ..., s_{k-1}.

Condiţia de găsire a unei soluţii. În acest caz orice vector cu n componente care respectă restricţiile este o soluţie. Deci dacă $k = n$ înseamnă că a fost găsită o soluţie.

Prelucrarea soluţiilor. Fiecare soluţie obţinută va fi afişată.

Întrucât $A_k = \{1, \ldots, n\}$, se poate considera că $s_k = i_k$. Cu aceste remarci algoritmul de generare a permutărilor poate fi descris ca în 9.2.

9.1.1 Varianta recursivă a algoritmului

Datorită faptului că după completarea componentei k problema se reduce la una similară de completare a componentei $k + 1$ cu una dintre valorile valide, tehnica backtracking poate fi uşor descrisă recursiv. Considerând ca parametru al algoritmului numărul de ordine al componentei care trebuie completată în cadrul apelului curent şi considerând că celelalte date (n, mulţimile A_1,...,A_n) au caracter global, algoritmul poate fi descris ca în 9.3.

Algoritmul recursiv se apelează pentru $k = 1$ (completarea soluţiei începe cu prima componentă): *backtracking_recursiv*(1). Pentru problema generării permutărilor de ordin n varianta recursivă a algoritmului este descrisă în 9.4.

9.2 Aplicaţii ale tehnicii căutării cu revenire

9.2.1 Generarea tuturor submulţimilor unei mulţimi

Se consideră problema determinării tuturor celor 2^n submulţimi ale unei mulţimi $X = \{x_1, x_2, \ldots, x_n\}$. Orice submulţime $S \subset X$ poate fi descrisă prin vectorul

Algoritmul 9.2 Generarea permutărilor folosind tehnica căutării cu revenire

```
permutări(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid=falseand  (s[k] ≤ n)
  do
    if validare(s[1..k]) =true then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
validare(s[1..k])
for i ← 1, k − 1 do
  if s[k] = s[i] then
    return  false
  end if
end for
return  true
```

Algoritmul 9.3 Varianta recursivă a căutării cu revenire

```
backtracking_recursiv(k)
if (s_1, ..., s_{k−1}) este soluţie then
  "s-a gasit o soluţie" // condiţia de ieşire
else
  for j ← 1, m_k do
    s_k ← a_j^k // se completează componenta k
    if (s_1, ..., s_k) "satisface condiţiile de continuare" then
      // se trece la completarea următoarei componente
      backtracking_recursiv(k + 1)
    end if
  end for
end if
```

Algoritmul 9.4 Varianta recursivă pentru generarea permutărilor

```
permutari_recursiv(k)
if k = n + 1 then
  afisare(s[1..k])
else
  for j ← 1, n do
    s[k] ← j
    if validare(s[1..k])=true then
      permutari_recursiv(k + 1)
    end if
  end for
end if
```

caracteristic $s = (s_1, \ldots, s_n)$ caracterizat prin:

$$s_k = \begin{cases} 1 & \text{dacă } x_k \in S \\ 0 & \text{dacă } x_k \notin S \end{cases}$$

Astfel $A_1 = \ldots = A_n = \{0, 1\}$ şi orice vector cu componente 0 sau 1 este o soluţie validă (nu se impun restricţii). Rezultă că problema este echivalentă cu a genera elementele produsului cartezian $A_1 \times \ldots \times A_n = \{0, 1\}^n$. Algoritmul poate fi descris ca în 9.5.

Algoritmul 9.5 Generarea tuturor submulţimilor unei mulţimi

```
submulţimi(n)
k ← 1
s[k] ← −1
while k > 0 do
  s[k] ← s[k] + 1
  if s[k] ≤ 1 then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← −1
    end if
  else
    k ← k − 1
  end if
end while
```

```
subm_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  s[k] ← 0; subm_rec(k + 1)
  s[k] ← 1; subm_rec(k + 1)
end if
```

Pornind de la acest algoritm poate fi descris cel pentru generarea tuturor submulţimilor cu m elemente (*combinări de n luate câte m*) ale unei mulţimi $A = \{a_1, \ldots, a_n\}$. Această problemă se caracterizează prin faptul că soluţiile

satisfac restricţia că numărul de componente egale cu 1 (sau echivalent suma tuturor componentelor) este egală cu m ($\sum_{j=1}^{n} s_j = m$). Această restricţie conduce la condiţia de continuare: $\sum_{j=1}^{k} s_j \leq m$. Un vector $(s_1, \ldots, s_k)$ este soluţie dacă $\sum_{j=1}^{k} s_j = m$. Prin modificarea algoritmilor pentru generarea tuturor submulţimilor se obţin variantele din Algoritmul 9.6.

Algoritmul 9.6 Generarea tuturor submulţimilor cu m elemente

```
combinări(n,m)
k ← 1
s[k] ← −1
while k > 0 do
  s[k] ← s[k] + 1
  if (s[k] ≤ 1) and suma(s[1..k]) ≤ m
  then
    if suma(s[1..k]) = m then
      afisare(s[1..k])
    else
      if k < n then
        k ← k + 1; s[k] ← −1
      end if
    end if
  else
    k ← k − 1
  end if
end while
```

```
comb_rec(k)
if suma(s[1..k − 1]) = m then
  afisare(s[1..k − 1])
else
  if k <= n then
    s[k] ← 0; comb_rec(k + 1)
    s[k] ← 1; comb_rec(k + 1)
  end if
end if
```

Algoritmul `suma`$(s[1..k])$ calculează suma elementelor din $s[1..k]$ iar algoritmul `afisare` apelat pentru $s[1..k]$ afişează afişează elementele a_i pentru care $s[i] = 1$. Se observă că vectorii având toate componentele completate ($k = n$) dar pentru care suma elementelor este diferită de m sunt ignoraţi.

9.2.2 Amplasarea damelor pe tabla de şah

O problemă clasică (enunţată de Gauss în 1850) de generare a unor configuraţii ce respectă anumite restricţii este cea a amplasării damelor pe o tablă de şah astfel încât să nu se atace reciproc. Considerăm cazul general în care n dame trebuie amplasate în cadrul unei matrici pătratice $n \times n$ astfel încât pe nici o linie, pe nici o coloană şi pe nici o diagonală să nu se afle două dame.
Reprezentarea soluţiei. Întrucât pe fiecare linie se va afla exact o damă şi toate damele sunt identice este suficient să reţinem coloana pe care se află fiecare dintre ele. Astfel soluţia problemei va fi de forma: $(s_1, s_2, \ldots, s_n)$ unde s_k indică coloana pe care se va afla dama de pe linia k. Vectorii corespunzători configuraţiilor din figura 9.2 sunt $(3, 1, 4, 2)$ respectiv $(2, 4, 1, 3)$. Pentru valori

mai mari ale lui n numărul configuraţiilor devine din ce în ce mai mare (pentru $n = 8$ există 92 de configuraţii).

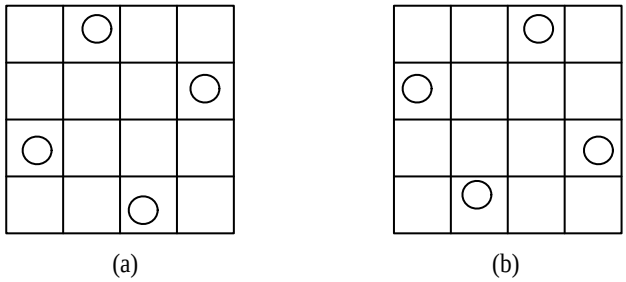

Figura 9.2: Configuraţii valide pentru problema damelor ($n = 4$)

Restricţii şi condiţii de continuare. Din modul de reprezentare a soluţiilor restricţia ca damele să nu fie plasate pe aceeaşi linie este implicit satisfăcută. Condiţia ca pe orice coloană să se afle o singură damă este echivalentă cu $s_i \neq s_j$ pentru orice $i \neq j$. În ceea ce priveşte condiţia referitoare la diagonale pornim de la observaţia că două elemente ale unei matrici aflate pe poziţiile (i_1, j_1) respectiv (i_2, j_2) se află pe aceeaşi diagonală dacă $i_1 - j_1 = i_2 - j_2$ sau $i_1 + j_1 = i_2 + j_2$. Astfel condiţia ca două dame să nu se afle pe aceeaşi diagonală este: $i - s_i \neq j - s_j$ şi $i + s_i \neq j + s_j$ pentru orice $i \neq j$. Cele două relaţii sunt echivalente cu $|i - j| \neq |s_i - s_j|$ pentru orice $i \neq j$. La completarea componentei k condiţiile de continuare sunt prin urmare: $s_k \neq s_i$ şi $|k - i| \neq |s_k - s_i|$ pentru orice $k \neq i$.
Descrierea algoritmului. Algoritmul este descris în 9.7 atât în variantă iterativă cât şi în variantă recursivă. Se observă că structura algoritmului este cea generală, pentru $A_1 = A_2 = \ldots = A_n = \{1, \ldots, n\}$ iar funcţia `validare` implementează condiţiile de continuare.

9.2.3 Colorarea hărţilor

Se consideră o hartă cu n ţări care trebuie colorată folosind $m < n$ culori, astfel încât oricare două ţări vecine să fie colorate diferit. Relaţia de vecinătate dintre ţări este reţinută într-o matrice $n \times n$ ale cărei elemente sunt:

$$v_{ij} = \begin{cases} 1 & \text{dacă } i \text{ e vecină cu } j \\ 0 & \text{dacă } i \text{ nu e vecină cu } j \end{cases}$$

Reprezentarea soluţiilor. O soluţie a problemei este o modalitate de colorare a ţărilor şi poate fi reprezentată printr-un vector $(s_1, \ldots, s_n)$ cu $s_i \in \{1, \ldots, m\}$ reprezentând culoarea asociată ţării i. Mulţimile de valori ale elementelor sunt $A_1 = \ldots = A_n = \{1, \ldots, m\}$.

Algoritmul 9.7 Amplasarea damelor

```
plasare_dame(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid =false) and (s[k] ≤
  n) do
    if  validare(s[1..k])  =true
    then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
dame_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  for i ← 1, n do
    s[k] ← i
    for validare(s[1..k]) do
      dame_rec(k + 1)
    end for
  end for
end if
validare(s[1..k])
for i ← 1, k − 1 do
  if (s[k] = s[i]) or (|k − i| = |s[k] −
  s[i]|) then
    return  false
  end if
end for
return  true
```

Restricţii şi condiţii de continuare. Restricţia ca două ţări vecine să fie colorate diferit se specifică prin: $s_i \neq s_j$ pentru orice i şi j având proprietatea $v_{ij} = 1$. Condiţia de continuare pe care trebuie să o satisfacă soluţia parţială $(s_1, \ldots, s_k)$ este: $s_k \neq s_i$ pentru orice $i < k$ cu proprietatea că $v_{ik} = 1$.
Descrierea algoritmului. Variantele iterativă şi recursivă sunt descrise în 9.8 în care este folosit următorul algoritm de validare.

```
validare(s[1..k])
for i ← 1, k − 1 do
  if (s[k] = s[i]) and (v[i, k] = 1) then
    return  false
  end if
end for
return  true
```

Algoritmul 9.8 Colorarea hărţilor

```
colorare(n)
k ← 1
s[k] ← 0
while k > 0 do
  s[k] ← s[k] + 1
  valid ← false
  while (valid =false) and (s[k] ≤ m) do
    if validare(s[1..k]) =true then
      valid ← true
    else
      s[k] ← s[k] + 1
    end if
  end while
  if valid =true then
    if k = n then
      afisare(s[1..n])
    else
      k ← k + 1; s[k] ← 0
    end if
  else
    k ← k − 1
  end if
end while
```

```
colorare_rec(k)
if k = n + 1 then
  afisare(s[1..n])
else
  for i ← 1, m do
    s[k] ← i
    if validare(s[1..k]) then
      colorare_rec(k + 1)
    end if
  end for
end if
```

9.2.4 Determinarea tuturor drumurilor dintre două oraşe

Se consideră o mulţime de n oraşe $\{o_1, o_2, \ldots, o_n\}$ şi o matrice binară cu n linii şi n coloane care specifică între care oraşe există drumuri directe. Elementele matricii sunt de forma:

$$v_{ij} = \begin{cases} 1 & \text{dacă există drum direct între } i \text{ şi } j \\ 0 & \text{altfel} \end{cases}$$

Se pune problema determinării tuturor drumurilor care leagă oraşul o_p de oraşul o_q.

Reprezentarea soluţiilor. Spre deosebire de problemele anterioare nu toate soluţiile au aceeaşi lungime. O soluţie a problemei este de forma $(s_1, \ldots, s_m)$ cu $s_i \in \{1, 2, \ldots, n\}$ indicând oraşul care va fi parcurs în etapa i a traseului.

Restricţii şi condiţii de continuare. O soluţie $(s_1, \ldots, s_m)$ trebuie să satisfacă: $s_1 = p$ (oraşul de start), $s_m = q$ (oraşul destinaţie), $s_i \neq s_j$ pentru orice $i \neq j$ (nu se trece de două ori prin acelaşi oraş), $v_{s_i s_{i+1}} = 1$ pentru $i = \overline{1, n-1}$ (între oraşele parcurse succesiv există drum direct). La completarea componentei

k, condiţia de continuare este $s_k \neq s_i$ pentru $i = \overline{1, k-1}$ şi $v_{s_{k-1}s_k} = 1$. Condiţia de găsire a unei soluţii nu este determinată de numărul de componente completate ci de faptul că $s_k = q$.

Varianta recursivă şi funcţia de validare sunt descrise în Algoritmul 9.9.

Algoritmul 9.9 Determinarea tuturor traseelor între două oraşe

```
drumuri_rec(k)
if s[k-1] = q then
  afisare(s[1..k-1])
else
  if k ≠ n+1 then
    for i ← 1, n do
      s[k] ← i
      if  validare(s[1..k])   =true
      then
        drumuri_rec(k+1)
      end if
    end for
  end if
end if
```

```
validare(s[1..k])
if v[s[k-1], s[k]] = 0 then
  return false
else
  for i ← 1, k-1 do
    if (s[k] = s[i]) then
      return false
    end if
  end for
end if
return true
```

Înainte de apelul algoritmului *drumuri_rec* se completează $s[1]$ cu p iar la apel se specifică: *drumuri_rec*(2).

9.3 Tehnica căutării de tip "ramifică şi mărgineşte"

Căutarea cu revenire permite parcurgerea sistematică a spaţiului soluţiilor şi abandonarea căilor care nu conduc la soluţii fezabile. În cazul problemelor de optimizare cu restricţii abandonarea unei ramuri în arborele de parcurgere a spaţiului soluţiilor se poate face nu doar când sunt încălcate restricţiile ci şi atunci când se poate decide că urmând calea respectivă nu este posibil să se obţină o configuraţie mai bună decât cea descoperită până la etapa curentă. Aceasta este ideea tehnicii de căutare bazată pe "ramificare şi mărginire" (branch and bound).

Să considerăm problema determinării unei configuraţii $s = (s_1, s_2, \ldots, s_n) \in S$ care satisface anumite restricţii şi care optimizează o funcţie obiectiv, $f : S \to \mathbb{R}$. La fel ca în cazul tehnicii backtracking soluţia se construieşte succesiv prin completarea componentelor iar la fiecare etapă se verifică dacă soluţia parţială $(s_1, s_2, \ldots, s_k)$ este *promiţătoare*. O soluţie parţială, $s_{(k)} = (s_1, s_2, \ldots, s_k)$, este considerată promiţătoare dacă: (i) nu încalcă restricţiile problemei; (ii) există şansa ca prin completarea celorlalte componente să se ajungă la o configuraţie mai bună decât ceea ce s-a obţinut până în momentul curent al parcurgerii.

La fel ca şi în cazul tehnicii backtracking procesul de construire a soluţiei poate fi ilustrat utilizând un arbore de decizie caracterizat prin faptul că nivelul k corespunde completării componentei k a soluţiei. Dacă nodului curent îi corespunde soluţia parţială $(s_1, \ldots, s_k)$ atunci se parcurg următorii paşi:

Ramificare. Se construiesc toate soluţiile parţiale $(s_1, \ldots, s_k, s_{k+1})$ prin completarea succesivă a poziţiei $k+1$ cu toate valorile posibile.

Mărginire. Pentru fiecare dintre soluţiile parţiale construite în pasul anterior se verifică dacă satisface restricţiile problemei. Pentru soluţiile ce satisfac restricţiile se estimeaza margini (inferioare şi/sau superioare) corespunzătoare valorii funcţiei obiectiv. Calculul acestor margini depinde de problema de rezolvat şi ele sunt folosite pentru a estima şansa ca o soluţie parţială să conducă la o soluţie mai bună decât cea mai bună soluţie construită până în momentul curent.

In cazul unei probleme de maximizare a doua proprietate specifică faptul că marginea superioară a funcţiei obiectiv a unei configuraţii având primele k componente egale cu $s_{(k)}$ adică $f^*(s_{(k)}) = \max_{(x_{k+1},\ldots,x_n)} f(s_1, \ldots, s_k, x_{k+1}, \ldots, x_n)$ nu este mai mică decât cea mai mare valoare a funcţiei obiectiv corespunzătoare configuraţiilor complete construite până în etapa curentă. In cazul unei probleme de minimizare se estimează marginea inferioară a funcţiei obiectiv şi se urmăreşte ca aceasta să nu fie mai mare decât valoarea asociată celei mai bune soluţii descoperite până în etapa curentă.

O soluţie parţială este abandonată fie dacă nu satisface restricţiile (ca în cazul tehnicii backtracking) fie dacă marginea superioară a funcţiei obiectiv este mai mică decât cea mai bună valoare a unei soluţii (în cazul unei probleme de maximizare) sau dacă marginea inferioară este mai mare decât cea mai bună valoare (în cazul unei probleme de minimizare).

Exemplul 9.3 (*Problema rucsacului*) Reluăm problema selectării unui subset de obiecte dintre n obiecte având dimensiunile $d_1,\ldots,d_n$ şi valorile $v_1,\ldots,v_n$ astfel încât suma dimensiunilor obiectelor selectate să nu depăşească capacitatea C a rucsacului iar suma valorilor să fie maximă. Dacă reprezentăm un subset de obiecte printr-un vector $(s_1, \ldots, s_n) \in \{0,1\}^n$ cu proprietatea că $s_k = 1$ înseamnă că obiectul k a fost selectat iar $s_k = 0$ înseamnă că obiectul nu a fost selectat atunci restricţia problemei se exprimă: $\sum_{i=1}^n s_i d_i \leq C$ iar funcţia obiectiv este $V(s_1, \ldots, s_n) = \sum_{i=1}^n s_i v_i$.

Fiind o problemă de maximizare fiecărei soluţii parţiale $(s_1, \ldots, s_k)$ i se poate asocia ca margine superioară:

$$V^*(s_1, \ldots, s_k) = \sum_{i=1}^{k} s_i v_i + (C - \sum_{i=1}^{k} s_i d_i) \max_{i=\overline{k+1,n}} v_i/d_i \tag{9.1}$$

care reprezintă valoarea care ar fi obţinută dacă zona liberă din rucsac ar fi umplută cu cel mai valoros obiect neselectat încă. Această margine este în

general una largă, în sensul că nu întotdeauna este atinsă. Să considerăm următorul exemplu concret: $C = 4$, $d = (1, 2, 3, 2)$, $v = (8, 14, 15, 10)$. Valorile relative ale obiectelor sunt: $p = (8, 7, 5, 5)$. Considerăm că obiectele sunt ordonate descrescător după valoarea relativă. Acest lucru nu este esenţial dar poate facilita identificarea mai rapidă a unei prime soluţii candidat ce poate fi utilizată ulterior ca referinţă. Se observă că aplicarea tehnicii greedy conduce la soluţia $(1, 1, 0, 0)$ corespunzătoare selectării primelor două obiecte având valoarea totală egală cu 22.

Arborele de decizie corespunzător parcurgerii spaţiului soluţiilor este ilustrat în figura 9.3. Fiecare nod conţine subsetul de obiecte deja selectat, dimensiunea spaţiului încă disponibil în rucsac (C_r) şi marginea superioară a valorii totale (V^*). Abandonarea unei ramuri din cauza încălcării restricţiei (nu există suficient spaţiu liber în rucsac) este marcată cu **X** iar abandonarea unui nod din cauză că marginea superioară este mai mică decât cea asociată celei mai bune soluţii descoperite (V_B) este marcată printr-o linie orizontală. Valoarea V_B este actualizată ori de câte ori se ajunge la o soluţie mai bună.

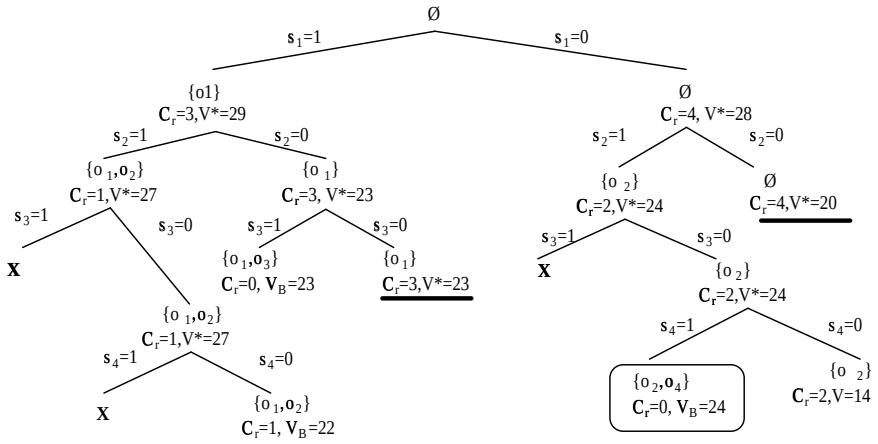

Figura 9.3: Arborele de decizie utilizat în rezolvarea problemei rucsacului folosind tehnica "ramifică şi mărgineşte"

Etapele parcurse în proiectarea unui algoritm bazat pe tehnica branch and bound sunt similare celor de la tehnica backtracking:

- Se alege reprezentarea soluţiei şi se identifică mulţimile de valori corespunzătoare componentelor soluţiei. În cazul problemei rucsacului mulţimile A_k sunt toate egale cu $\{0, 1\}$.
- Se deduc condiţiile de continuare a ramificării (restricţiile pe care trebuie să le satisfacă soluţiile parţiale). Pentru problema rucsacului condiţia de

continuare corespunzătoare unei soluţii parţiale $(s_1, \ldots, s_k)$ este $\sum_{i=1}^{k} s_i d_i < C$.

- Se stabileşte modul de calcul al marginilor corespunzătoare funcţiei obiectiv. Este elementul specific tehnicii "ramifică şi mărgineşte" iar în cazul problemei rucsacului se bazează pe relaţia 9.1.
- Se stabileşte ordinea de parcurgere a soluţiilor parţiale obţinute prin ramificarea unui nod. Nodurile corespunzătoare unui nivel sunt ramificate fie în ordinea în care au fost obţinute fie descrescător după valoarea limitei superioare (în cazul problemelor de maximizare) sau crescător după valoarea limitei inferioare (în cazul problemelor de minimizare). Această strategie nu garanteaza întotdeauna faptul că soluţia va fi descoperită mai devreme (după cum se observă din figura 9.3 soluţia se obţine nu prin ramificarea nodului având $V^* = 29$ ci a celui cu $V^* = 28$).
- Se stabileşte modul în care se decide dacă s-a obţinut o soluţie. În cazul problemei rucsacului s-a ajuns la o soluţie fezabilă fie dacă nu mai există alte obiecte de selectat fie dacă a fost acoperită întreaga capacitate a rucsacului. O dată cu obţinerea unei soluţii fezabile se actualizează V_B, dacă valoarea noii soluţii este mai bună decât V_B. Pentru a se putea efectua comparaţiile V_B trebuie iniţializată (în cazul unei probleme de maximizare cu o valoare cât mai mică, iar in cazul uneia de minimizare cu o valoare cât mai mare). În cazul problemei rucsacului V_B se poate iniţializa cu 0.

  O dată cu stocarea valorii în V_B se stochează si soluţia corespunzătoare în S_B. La sfârşit soluţia problemei se va găsi în S_B.

Structura generală a unui algoritm bazat pe tehnica "ramifică şi mărgineşte" este descris în 9.10. Descrierea se bazează pe următoarele ipoteze: (i) se rezolvă o problemă de maximizare; (i) S_B şi V_B sunt variabile globale iar V_B este iniţializată cu o valoare suficient de mică; (iii) algoritmul se apelează pentru $k = 1$ iar rezultatul se colectează în S_B.

Principalele diferenţe între tehnica backtracking şi branch and bound sunt:

- Tehnica backtracking se aplică în general pentru problemele de satisfacere a restricţiilor pe când tehnica branch and bound se foloseşte pentru rezolvarea problemelor de optimizare.
- În cazul tehnicii backtracking abandonarea unei ramuri se face doar când sunt încalcate restricţiile pe când în cazul tehnicii branch and bound se face ori de câte ori se ajunge la o configuraţie care nu este promiţătoare (nu este posibil să conducă la o configuraţie pentru care valoarea funcţiei obiectiv să fie mai bună decât cea corespunzătoare soluţiei optime curente).

Algoritmul 9.10 Structura generală a unui algoritm bazat pe tehnica "ramifică şi mărgineşte"

```
BB(k)
if (s_1, ..., s_{k-1}) este soluţie then
   if V(s_1, ..., s_{k-1}) > V_B then
      S_B ← (s_1, ..., s_{k-1}); V_B ← V(s_1, ..., s_{k-1})
   end if
else
   for j ← 1, m_k do
      s_k ← a_j^k
      if (s_1, ..., s_k) satisface condiţiile de continuare then
         calcul valoare limită V*(s_1, ..., s_k)
      end if
   end for
   ordonează soluţiile parţiale în funcţie de valorile limită
   for toate soluţiile fezabile do
      if V*(s_1, ..., s_k) > V_B then
         BB(k + 1)
      end if
   end for
end if
```