Divide et Impera, Backtracking, Greedy

Asist. Horea Mureșan

24 Februarie 2024

1 Introducere

- Divide et Impera (divide and conquer) - clasă de algoritmi care împarte recursiv o problemă în mai multe subprobleme similare până când acestea pot fi rezolvate direct. Acești algoritmi sunt eficienți la probleme de sortare, înmulțirea numerelor mari (algoritmul Karatsuba), calculul tranformării Fourier (procesare de semnale audio) etc.
- Backtracking - clasă de algoritmi folosită pentru identificarea soluțiilor prin "forță brută". Util în rezolvarea problemelor de satisfacere a constrângerilor, cum ar fi problema rucsacului, problema celor 8 regine, Sudoku. Este de asemenea folosit de unele limbaje de programare ca strategie de execuție (ex. Prolog).
- Greedy - clasă de algoritmi care implică alegerea optimă locală la fiecare pas al construirii soluției. Nu garantează furnizarea unei soluții optime global, dar aproximează o soluție locală optimă într-un timp relativ scurt.

2 Divide et Impera

2.1 Considerente teoretice

Algoritmii de tip divide et impera în general au trei părți:

- Divizarea problemei inițiale în subprobleme de aceeași natură, astfel încât procesul de divizare să poată fi aplicat (dacă este nevoie) din nou pe fiecare subproblemă
- Când subproblemele ajung la o dimensiune suficient de redusă, ele vor fi rezolvate direct. Decizia a ce reprezintă o dimensiune suficient de redusă depinde de problemă. Ca exemplu, pentru algoritmul de merge sort, se urmărește fragmentarea listei de sortat în subliste cu câte un element fiecare.
- După rezolvarea directă a subproblemelor, ultimul pas este combinarea soluțiilor mici pentru a construi soluția la problema inițială.

În pseudocod, structura generală a unui algoritm de tipul Divide et Impera este:

Divide et Impera:

```
1: function DIVIDE_ET_IMPERA(Problema)
2:     if Problema este o problemă elementară then
3:         return Rezolva(Problema)
4:     else
5:         [P1, P2] ← Descompune(Problema)
6:         R1 ← Divide_et_Impera(P1)
7:         R2 ← Divide_et_Impera(P2)
8:         return Combina(R1, R2)
9:     end if
10: end function
```

2.2 Probleme

Turnurile din Hanoi:

```
1: function HANOI(n, Sursa, Destinatie, Aux)
2:     if n ≥ 1 then
3:         Hanoi(n − 1, Sursa, Aux, Destinatie)
4:         TransferaDisc(Sursa, Destinatie)
5:         Hanoi(n − 1, Aux, Destinatie, Sursa)
6:     end if
7: end function
```

În pseudocod, fie vectorul V pe care îl sortăm:

Merge sort:

```
1: procedure MERGESORT(V, Start, End)
2:     if Start == End then
3:         return
4:     end if
5:     mid ← (Start + End)/2
6:     mergeSort(V, Start, mid)
7:     mergeSort(V, mid + 1, End)
8:     merge(V, Start, End)
9: end procedure
1: procedure MERGE(V, Start, End)
2:     mid ← (Start + End)/2
3:     i ← Start
4:     j ← mid + 1
5:     k ← 1
6:     tmp ← vector cu End − Start + 1 poziții
7:     while i ≤ mid && j ≤ End do
8:         if V[i] ≤ V[j] then
9:             tmp[k++] ← V[i++]
10:        else
11:            tmp[k++] ← V[j++]
12:        end if
13:    end while
14:    while i ≤ mid do
15:        tmp[k++] ← V[i++]
16:    end while
17:    while j ≤ End do
18:        tmp[k++] ← V[j++]
19:    end while
20:    V[Start..End] = tmp[1..k − 1]
21: end procedure
```

Algoritmul MergeSort reprezentat vizual:

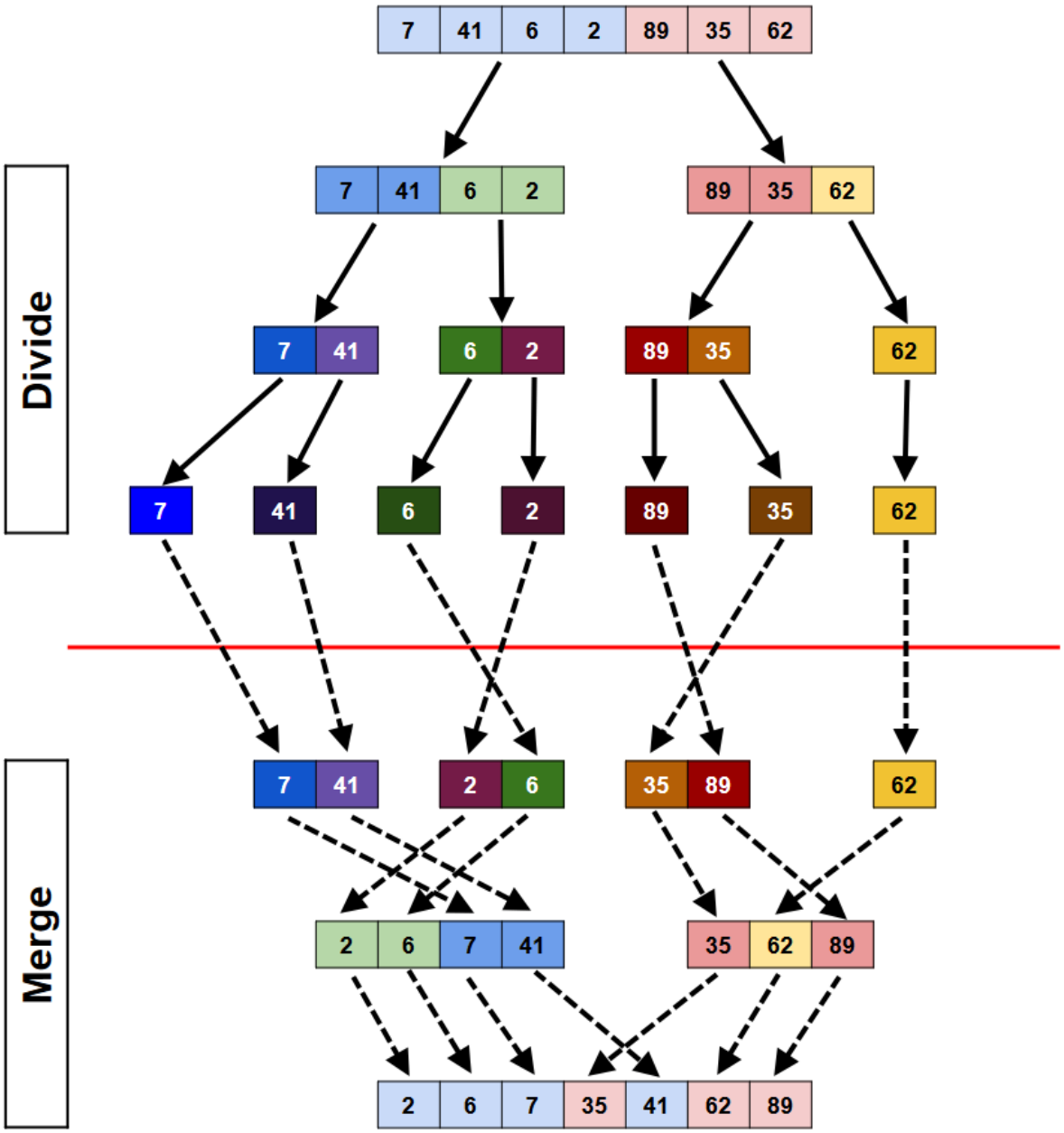

Fig. 1: Merge sort vizualizat

3 Backtracking

3.1 Considerente teoretice

Algoritmii de tip backtracking construiesc soluții incremental, eliminându-le automat pe cele care incalcă una sau mai multe constrângeri. Acești algoritmi pot fi aplicați pe probleme care admit conceptul candidat parțial pentru soluție. Formal, putem spune ca metoda backtracking se poate aplica dacă:

- Soluția poate fi reprezentată printr-un șir $x = (x[1], x[2], ..., x[n])$, fiecare element $x[i]$ aparținând unei mulțimi cunoscute A_i
- Fiecare mulțime A_i este finită și elementele se află într-o relație de ordine precizată. (De multe ori $A_1 = A_2 = \ldots = A_n$)

Algoritmul de tip backtracking construiește vectorul x (numit vector soluție) astfel:
Fiecare pas k, începând (de regulă) de la pasul 1, se prelucrează elementul curent x[k] al vectorului soluție:

1. $x[k]$ primește pe rând valori din mulțimea corespunzătoare A_k;
2. la fiecare pas se verifică dacă configurația curentă a vectorului soluție poate duce la o soluție finală – dacă valoarea lui $x[k]$ este corectă în raport cu $x[1], x[2], \ldots x[k-1]$:
   (a) dacă valoarea nu este corectă, elementul curent X[k] primește următoarea valoare din Ak sau revenim la elementul anterior x[k-1], dacă X[k] a primit toate valorile din A_k – backtrack;
   (b) dacă valoarea lui $x[k]$ este corectă (avem o soluție parțială), se verifică existența unei soluții finale a problemei:
      i. dacă configurația curentă a vectorului soluție x reprezintă soluție finală, o afișăm sau o salvăm; (depinde de tipul de problemă și de cerință)
      ii. dacă nu am identificat o soluție finală trecem la următorul element, $x[k+1]$, și reluăm procesul pentru acest element;

Pe măsură ce se construiește, vectorul soluție x reprezintă o soluție parțială a problemei. Când vectorul soluție este complet construit, avem o soluție finală a problemei.

În pseudocod, considerând că avem șirul x în care vom construi soluții și mulțimile A_k, alături de subalgoritmii:

- procedura $valid(K)$ validează condițiile impuse de problemă pentru soluția parțială $x[1], x[2], \ldots, x[K]$.
- procedura $solutie(K)$ verifică dacă soluția parțială $x[1], x[2], \ldots, x[K]$ îndeplinește condițiile pentru a fi soluție finală.

Structura generală a unui algoritm Backtracking este:

```
1:  function BACKTRACKING(K)
2:      for i ← 1, size(A_K) do
3:          if valid(K) then
4:              if solutie(K) then
5:                  X este soluție
6:              else
7:                  backtracking(K + 1)
8:              end if
9:          end if
10:     end for
11: end function
```

3.2 Probleme

Una din problemele clasice care poate fi rezolvată cu un algoritm de tip backtracking este problema rucsacului. Se dă o listă de obiecte care au o valoare și o greutate. Se dă de asemenea un rucsac cu o limită superioară de greutate. Se cere lista de obiecte cu valoare maximă care au suma greutăților mai mică sau egală cu limita dată de rucsac.

Pentru rezolvare, se pot căuta toate configurațiile posibile de obiecte și se reține cea mai bună găsită. Însă această abordare are potențialul de a genera multe configurații care depășesc limita de greutate. Aceste configurații pot fi eliminate de la primul pas când sunt detectate folosind backtracking.

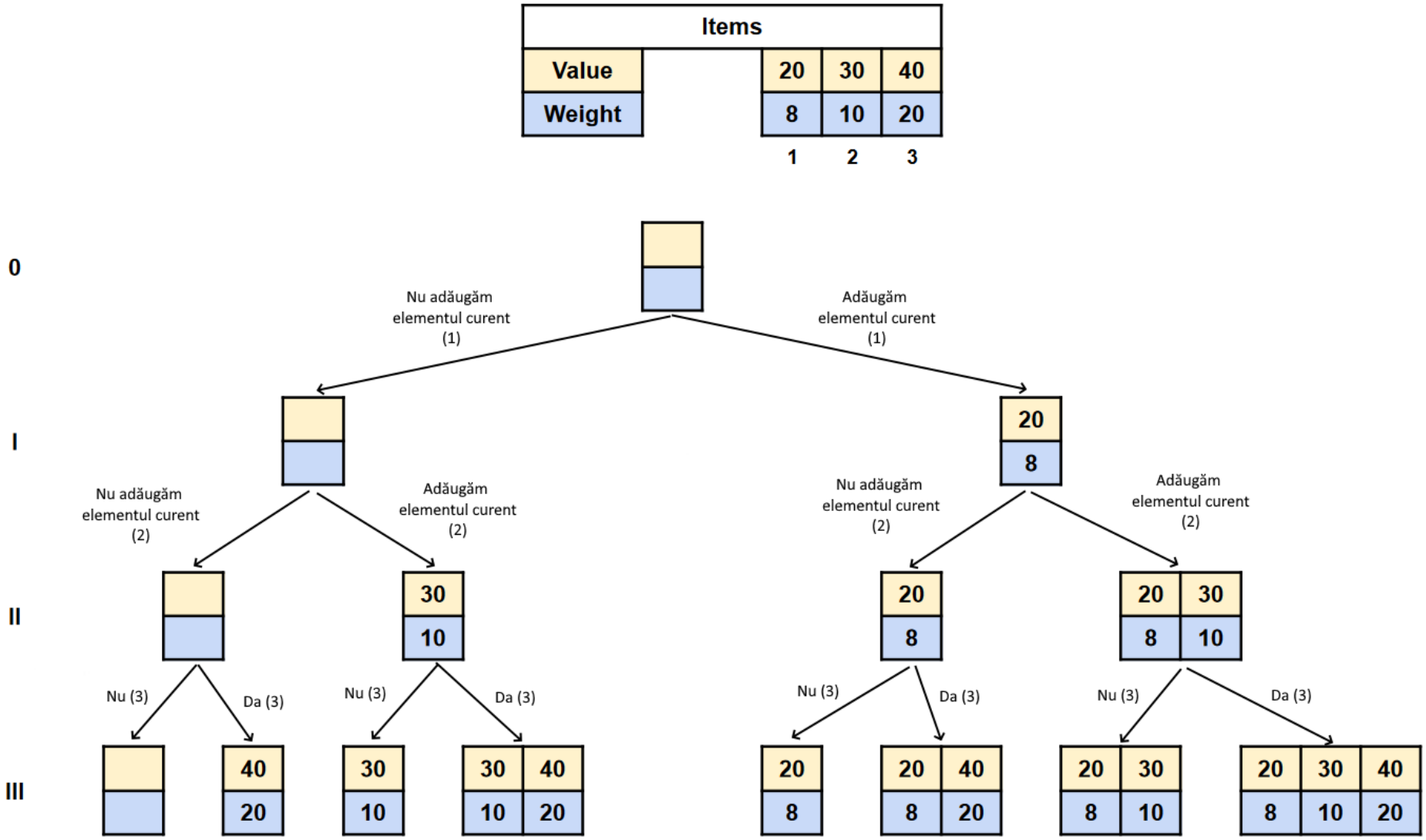

Fig. 2: Un algoritm de căutare exhaustivă explorează toate posibilitățile.

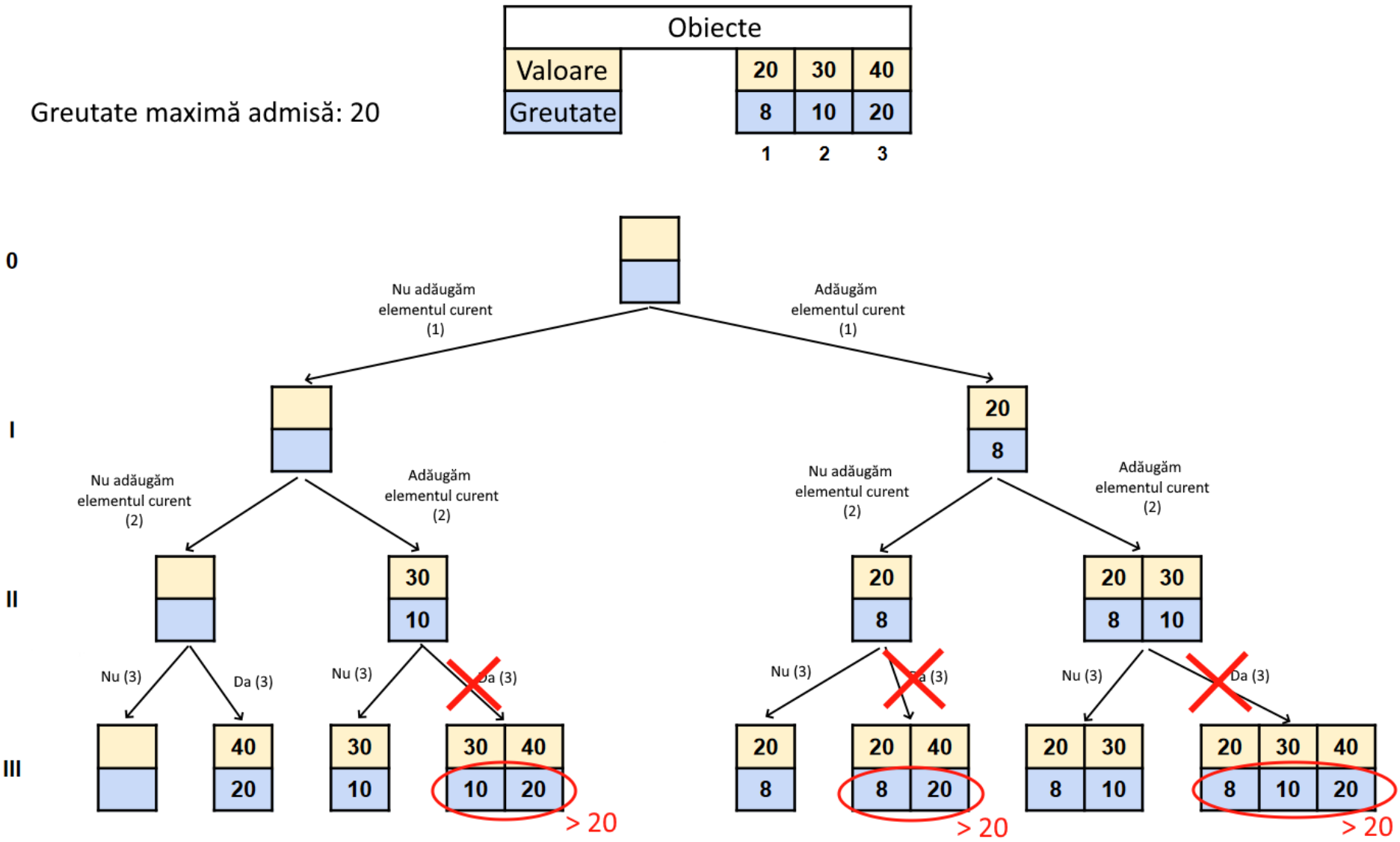

Fig. 3: Un algoritm de backtracking abandonează un candidat din momentul în care aceasta incalcă cel puțin o constrângere a problemei.

4 Greedy

4.1 Considerente teoretice

Algoritmii de tip Greedy încearcă să construiască rapid o soluție la o problemă (în general de optimizare) alegând la fiecare pas opțiunea care are imppactul cel mai mare la acel moment, fără a evalua calitatea globală a soluției. Aceste tipuri de rezolvări nu garantează că obțin soluții optime global, dar complexitatea lor este mică, și cu euristici suficient de bune, calitatea soluțiilor oferite nu este departe de cea optimă.

Structura generală a unui algoritm Greedy este:

```
1: function GREEDY(Candidates)
2:     Solution ← ∅
3:     while !solutie(Solution) && Candidates ≠ ∅ do
4:         X ← cel mai bun element din Candidates
5:         Candidates ← Candidates \ {X}
6:         if fezabil(Solution ∪ {X}) then
7:             Solution ← Solution ∪ {X}
8:         end if
9:     end while
10:    return Solution
11: end function
```

Se poate observa că la fiecare pas se alege cel mai bun candidat din lista de posibilități fără o analiză a viabilității acestuia după mai mulți pași. După alegerea unui candidat și includerea lui în soluție, acesta nu mai este scos din soluție, astfel că alternativele care nu includ acel candidat nu sunt explorate.

4.2 Probleme

Aplicând un algoritm de tip Greedy pentru problema rucsacului, putem ordona obiectele în ordine descrescătoare după valoarea lor. Asta asigură că cele mai valoroase obiecte vor fi incluse în soluție, dacă nu depășesc greutatea maximă. În Fig.4 observăm același exemplu ca la backtracking, în care varianta greedy nu găsește soluția optimă.

Pentru aceeași problemă, aplicăm o euristică puțin diferită. Sortăm obiectele în ordine descrescătoare după raportul valoare/greutate. Raționamentul este că, din moment ce avem o limită de greutate, vrem să alegem obiectele care maximizează valoarea per unitate de greutate. Nici această euristică nu garantează obținerea soluției optime global. De menționat, în cazul problemei continue a rucsacului (în care este permisă luarea unor fragmente din obiecte), această euristică obține tot timpul soluția optimă globală.

Algoritm Greedy pentru problema rucsacului:

```
1: function GREEDY(Obiecte)
2:     Solution ← ∅
3:     while Greutate_totala(Solution) ≤ GreutateMaxima && Obiecte ≠ ∅ do
4:         X ← obiectul cu raportul valoare/greutate maxim din Obiecte
5:         Obiecte ← Obiecte \ {X}
6:         if Greutate_totala(Solution ∪ {X}) ≤ GreutateMaxima then
7:             Solution ← Solution ∪ {X}
8:         end if
9:     end while
10:    return Solution
11: end function
```

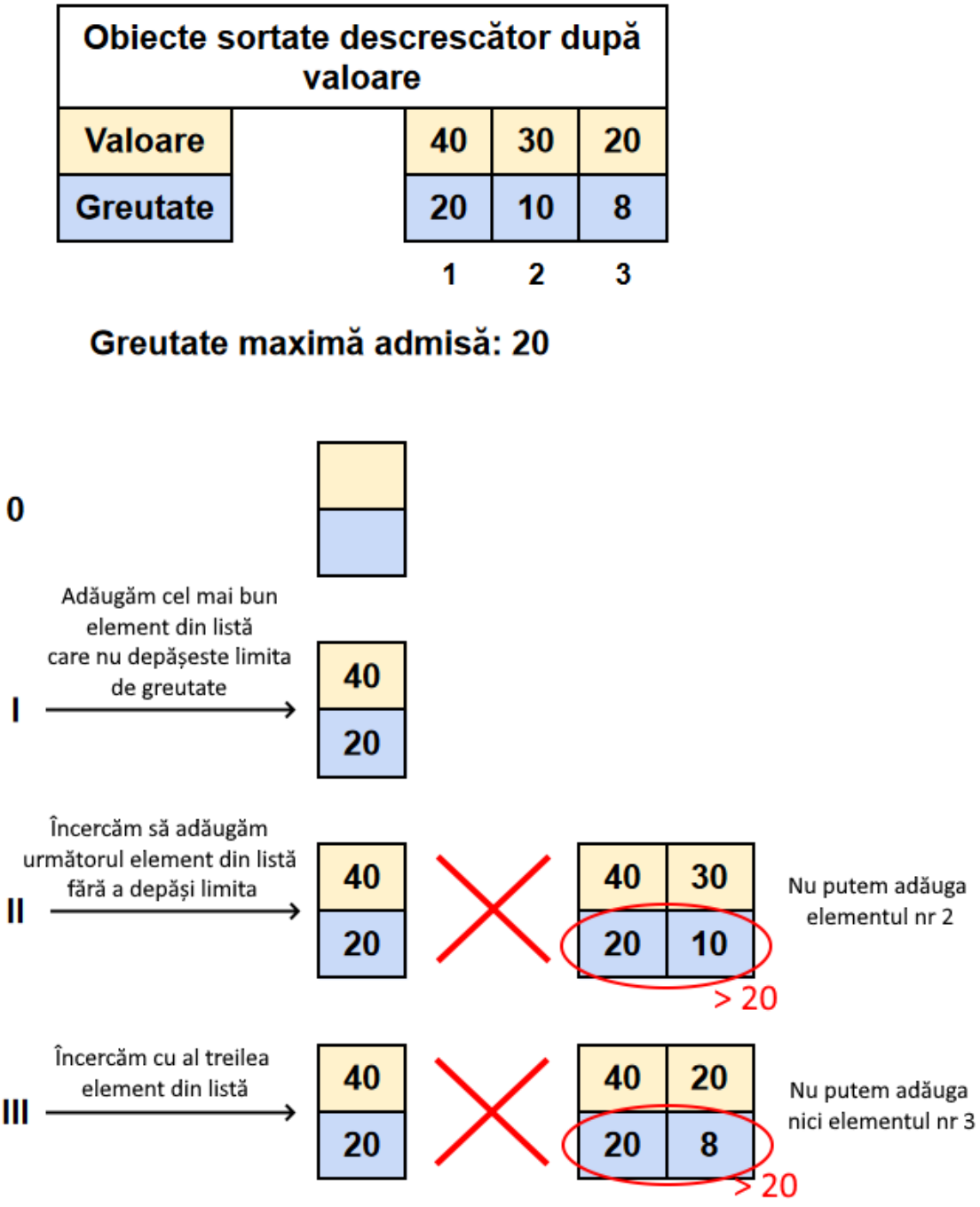

Fig. 4: Un exemplu în care euristica folosită nu este suficient de bună și soluția optimă nu este gasită.

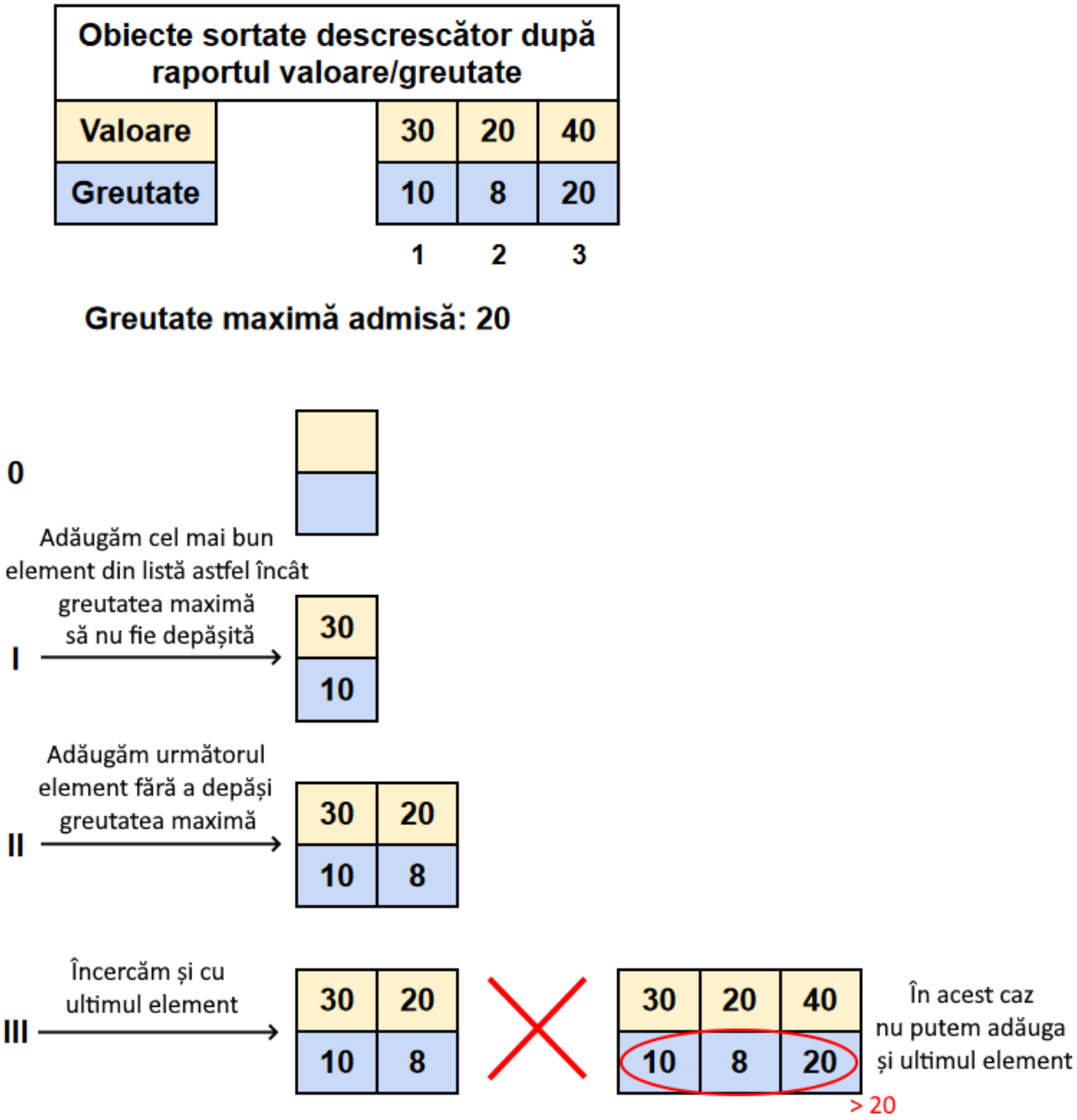

Fig. 5: În acest caz, euristica dată este suficient de bună și soluția optimă este găsită.