Tehnici de dezvoltare a algoritmilor (I)

Probleme combinatoriale. Cautare exhaustiva. Backtracking. Greedy. Metode euristice

Ex. problema: numararea restului (en. counting change)

- *Problema*: Un casier are la dispozitie o colectie de bancnote si monede de diferite valori. Se cere sa formeze o suma specificata folosind numarul minim de elemente
- Formulare matematica:
  - Se dau ***n*** bancnote si monede: ***P={p1, p2, …, pn}***
  - putem avea repetitii (2 monede de 50 bani, etc)
  - fie d_i valoarea lui p_i
  - Gasiti cea mai mica submultime S a lui P, S⊆P, astfel incat $\sum_{p_i \in S} d_i = A$

Numararea restului: cum arata o solutie?

- Reprezentam *S* ca o tupla de *n* valori: $X=\{x_1, x_2, \ldots, x_n\}$

$$\begin{cases} x_i = 1 & p_i \in S \\ x_i = 0 & p_i \notin S \end{cases}$$

- Fiind date $\{d_1, d_2, \ldots, d_n\}$, obiectivul este sa minimizam suma:

$$\sum_{i=1}^{n} x_i$$

- astfel incat:

$$\sum_{p_i \in S} d_i = A$$

Numararea restului: “brute force”

- Cautare exhaustiva, genereaza si testeaza
- Cautarea exhaustiva gaseste solutia optima prin enumerarea tuturor valorilor posibile pt. X:
  - Pentru fiecare valoare posibila a lui X, verificam constrangerea $\sum_{i=1}^{n} d_i x_i = A$
  - O valoare care satisface constrangerea se numeste *solutie fezabila*
  - Solutia - solutie fezabila care minimizeaza *functia obiectiv*:

$$\sum_{i=1}^{n} x_i$$

- Cate valori posibile avem pentru X? (dimensiunea spatiului de cautare)

Numararea restului: cautare exhaustiva

- Cautarea exhaustiva nu are o forma/structura specifica; se muleaza de regula pe enunt
- Pro: simplu de implementat, aplicabilitate larga
- Contra: nu e eficienta (exploreaza tot spatiul de cautare)
- Timpul de executie: $\Omega(n2^n)$
  - numarul de solutii candidat - $\Omega(2^n)$
  - estimarea fezabilitatii unei solutii: *O(n)*
  - calculul valorii functiei obiectiv: *O(n)*

Numararea restului: exemplu

- **A = 20**
- **D={1,1,1,1,1,10,10,15}**
- S:
  - X = {0,0,0,0,0,0,0,0}
  - X = {1,0,0,0,0,0,0,0}
  - …
  - X = {1,1,1,1,1,0,0,1} - *fezabila*
  - …
  - X = {0,0,0,0,0,1,1,0} - *fezabila, optima*
  - …
  - X = {1,1,1,1,1,1,1,1}

Alte exemple de algoritmi de tip forta bruta

- Calculul a^n (a>0, n - natural)
- Calcularea *n!*
- Inmultirea a 2 matrici
- Cautarea unui element intr-o lista
- Potrivire de string-uri
- Evaluare polinom in punctul x_0
- Gasirea celei mai apropiate perechi de puncte in plan
- TSP (Ciclu Hamiltonian)
- Problema rucsacului
- etc…

Brute force: Potrivire de string-uri

```
function NaiveSearch(string s[1..n], string pattern[1..m])
    for i from 1 to n-m+1
      for j from 1 to m
         if s[i+j-1] ≠ pattern[j]
            jump to next iteration of outer loop
        return i
    return not found
```

1) AAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH 5 comparatii
2) AAAAAAAAAAAAAAAAAAAAAAAAH
    AAAAH 5 comparatii
AAAAAAAAAAAAAAAAAAAH
   AAAAH 5 comparatii
.....
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
5 comparatii AAAAH

Eficienta?
O(mn) cazul defavorabil

Algoritmi mai eficienti?
- salt mai mare in caz de nepotrivire in for-ul exterior(Knuth–Morris–Pratt)
- bazat pe hashing (Robin Karp)

Brute force: Evaluare polinom

$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$, calculati $p(x_0)$

```
x = x0
p = 0.0
for i=n down to 0 do
    power = 1
    for  j = 1 to i do
       power =  power * x
     p = p + a[i] * power
return p
```

Eficienta: $O(n^2)$

```
x = x0
p = a[0]
power = 1
for i=1 to n do
      power =  power * x
      p = p + a[i] * power
return p
```

Eficienta: ?

Brute force: Gasirea celei mai apropiate perechi de puncte in plan

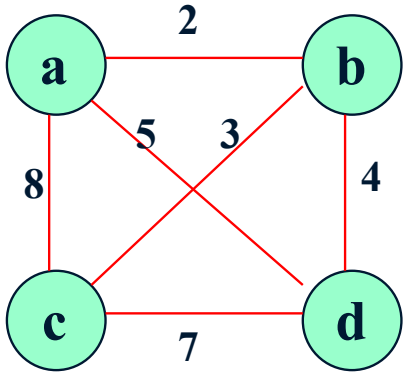
- Se calculeaza distanta intre oricare 2 puncte din plan, si se cauta minimul.
  - Eficienta?

Brute force: Gasirea celei mai apropiate perechi de puncte in plan

- Se calculeaza distanta intre oricare 2 puncte din plan, si se cauta minimul.
  - Eficienta: $O(n^2)$
- Pp. ca punctele sunt pe un segment de dreapta. Puteti da un algoritm mai bun?
  - se sorteaza punctele in ordine crescatoare
  - se calculeaza diferenta intre oricare 2 pcte adiacente
  - Se cauta cea mai mica astfel de diferenta
- Eficienta? ($O(nlgn)$)

Brute force: TSP - Traveling Salesman Problem

- Se dau *n* orase, cu distante cunoscute intre fiecare 2 dintre ele (i.e. *graf neorientat, complet, cu costuri pe muchii*), gasiti cel mai scurt tur care trece prin toate orasele o data, inainte de a reveni in orasul sursa.
- Alternativ: Gasiti cel mai scurt *Cicuit Hamiltonian* intr-un graf conex, cu ponderi



Brute force: TSP - Traveling Salesman Problem

Make a list of all possible Hamilton circuits (permutari posibile de noduri care incep si se termina cu nodul sursa)

Calculate the weight of each Hamilton circuit by adding up the weights of its edges

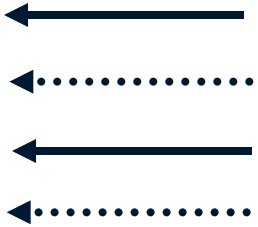
Choose the Hamilton circuit with the smallest total weight

Tur	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Efficienta?

Brute force: TSP - Traveling Salesman Problem

Tur	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17



Efficienta: (V-1)!/2 -> *O(V!)*

Daca un calculator ar calcula 1.000.000 circuite/sec

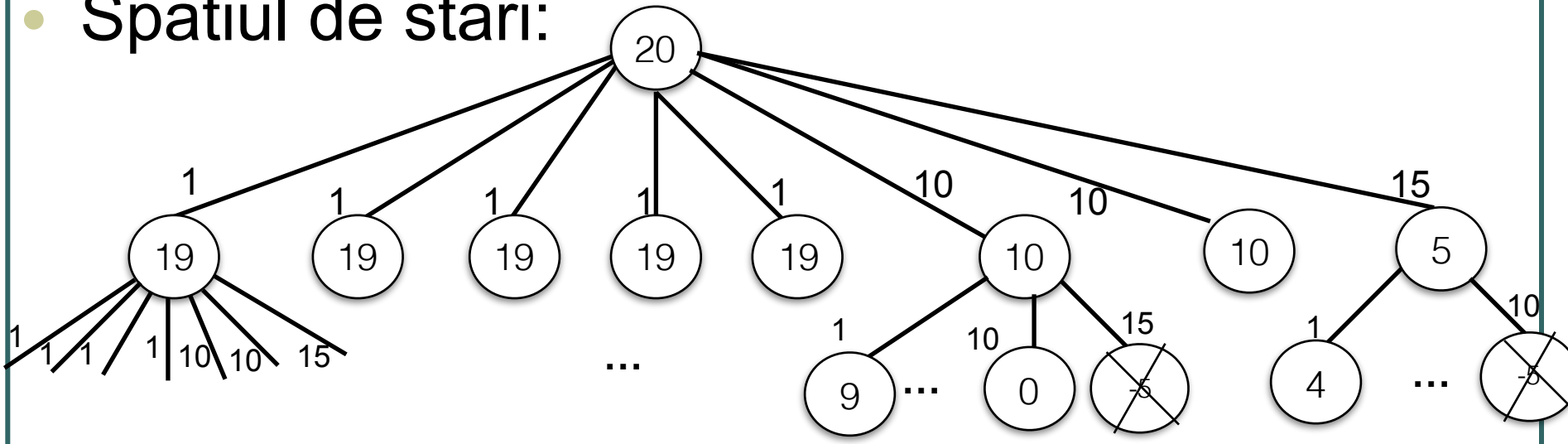
N = 6, 7, 8, 9: instantaneu
N = 10: ~ 1/3 sec
N = 11: ~ 4 sec
N = 12: ~ 40 sec
N = 13: ~ 8 min
N = 14: ~ 2 ore
N = 15: putin peste o zi
N = 20: peste 1.000.000 ani

Backtracking: Principiu fundamental

- Se parcurge “arborele de stari posibile” folosind o strategie de parcurgere in adancime
- Se utilizeaza cea mai buna solutie gasita pana in momentul curent pentru a elimina (en. *prune*) solutii partiale ne-promitatoare - i.e. care nu pot conduce la o solutie *fezabila* sau la o solutie *mai buna* decat cea gasita pana la acel moment
- Scopul este de a elimina suficiente stari din spatiul de cautare (*exponential* in marime), astfel incat solutia optima sa poata fi gasita intr-un timp rezonabil
- In cazul defavorabil, strategia da tot algoritmi exponentiali

Backtracking: Numararea restului

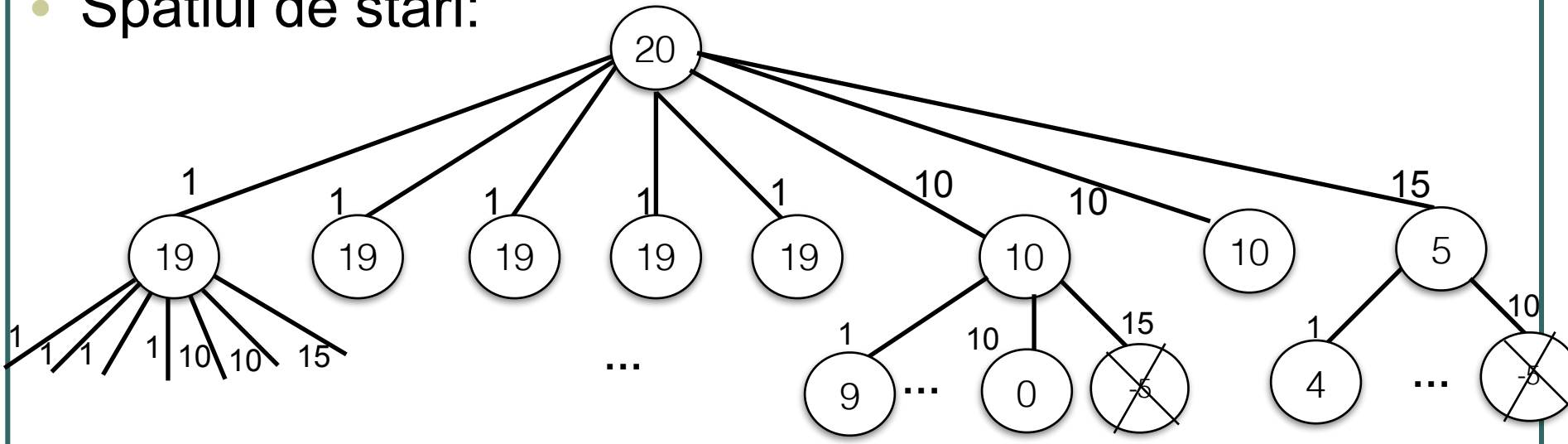
- Spatiul de stari:



- Care sunt criteriile de eliminare a starilor neinteresante?
- Cum e mai bine sa ordonam nodurile in arbore?

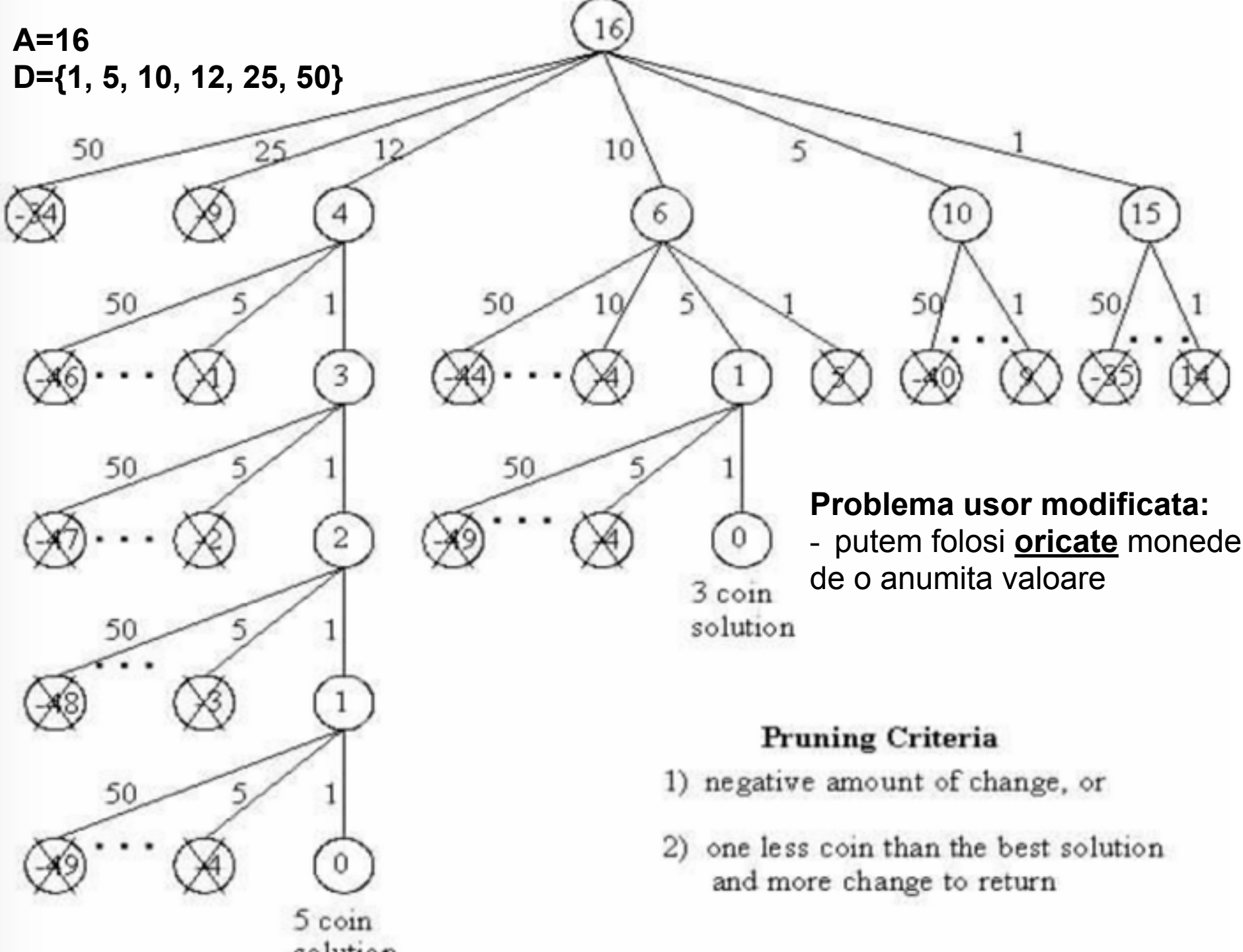
Backtracking: Numararea restului

- Spatiul de stari:



- Care sunt criteriile de eliminare a starilor neinteresante?
  - cantitate negativa a restului de returnat
  - la o singura moneda pana la cea mai buna solutie de pana atunci, mai avem rest de returnat
- Cum ordonam nodurile in arbore (pe acelasi nivel?)

A=16
D={1, 5, 10, 12, 25, 50}



Problema usor modificata:
- putem folosi **<u>oricate</u>** monede de o anumita valoare

Pruning Criteria

1) negative amount of change, or

2) one less coin than the best solution and more change to return

Backtracking: Numararea restului - problema modificata

```
void Backtrack(int numberOfCoins, int changeToBeReturned) {
     int i, j;
     for (i = 0; i < numberOfCoinTypes; i++) {
          partialSolution[i] = partialSolution[i] + 1;
          changeToBeReturned = changeToBeReturned - coinValues[i];
          numberOfCoins++;
          if (Promising(numberOfCoins, changeToBeReturned)) {
              if ((changeToBeReturned) == 0) {
                  if (!solutionFound || numberOfCoins < bestNumberOfCoins) {
                          // Found a new best solution, so save it
                         bestNumberOfCoins = numberOfCoins;
                         solutionFound = TRUE;
                         for (j = 0; j < numberOfCoinTypes; j++) {
                              bestSolutionSoFar[j] = partialSolution[j];
                         } // end for (j...
                  } // end if(!solution...
              } else {
                   Backtrack(numberOfCoins,changeToBeReturned);
              } // end if (changeToBeReturned...
           } // end if (promising...
           partialSolution[i] = partialSolution[i] - 1;
           changeToBeReturned = changeToBeReturned + coinValues[i];
           numberOfCoins--;
     } // end for (i...
} // end Backtrack
```

Backtracking: Numararea restului - problema modificata

```
int Promising(int numberOfCoins, int changeToBeReturned) {
     if (changeToBeReturned < 0) {
          return FALSE;
     } else if (solutionFound && changeToBeReturned > 0
                && numberOfCoins+1 >= bestNumberOfCoins) {
          return FALSE;
     } // end if
     return TRUE;
} // end Promising
```

Backtracking - Schema generala - pseudocod

```
Backtrack-DFS(A, k)
   if A  = (a1, a2, ..., ak) is a solution, report it
   else
     k  = k  + 1
     compute Sk
     while Sk != ∅  do
       ak  = an element in Sk
       Sk  = Sk – ak
       Backtrack-DFS(A, k)
```

La fiecare pas se incearca extinderea solutiei partiale prin adaugarea unui nou element, a_k
Daca dupa extindere avem o solutie, o procesam
Daca nu, vedem daca se poate extinde in continuare (generam in S_k starile urmatoare posibile, si pe exploram pe rand

Backtracking - construieste un arbore de solutii partiale - fiecare varf - solutie partiala; avem muchie de la *x* la *y* daca in *y* am ajuns extinzand solutia din *x*; arborele se parcurge in adancime

Backtracking - Schema generala - cod

```
bool finished = FALSE; /* found all solutions yet? */
backtrack(int a[], int k, data input)
{
  int c[MAXCANDIDATES]; /* candidates for next position */
  int ncandidates; /* next position candidate count */
  int i; /* counter */
  if (is_a_solution(a,k,input))
      process_solution(a,k,input);
  else {
      k = k+1;
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++){
          a[k] = c[i];
          make_move(a,k,input);
          backtrack(a,k,input);
          unmake_move(a,k,input);
          if (finished) return; /* terminate early */
      }
  }
}
```

Backtracking - Schema generala

- **`is_a_solution(a,k,input)`**- testeaza daca primele **`k`** elemente ale vectorului **`a`** formeaza o solutie; **`input`** ne permite sa transmitem informatie generala in functie (e.g. dimensiunea *n* a solutiei)
- **`construct_candidates(a,k,input,c,&ncandidates)`**– populeaza vectorul **`c`** cu multimea posibila de valori pentru pozitia **`k`** a lui **`a`**, cunoscandu-se primele **`k-1`** pozitii
- **`process solution(a,k,input)`** - afiseaza, numara, proceseaza o solutie completa odata de a fost construita
- **`make_move(a,k,input)`** si **`unmake_move(a,k,input)`** - ne permit sa modificam o structura in raspuns la ultima mutare efectuata (**`make_move`**), respectiv sa o curatam (**`unmake_move`**) daca decidem sa anulam mutarea; e mai eficient decat sa se reconstruiasca din *a* de fiecare data
- flag-ul **`finished`** permite terminarea prematura, si poate fi modificat in oricare din functiile de mai sus, in functie de necesitate (e.g. la gasirea unei solutii)
- ***TEMA***: Incercati sa identificati aceste functii pe codul de la algoritmul de numararea restului!

Backtracking - Generarea sub-multimilor

- 2^n -> solutia se repr. ca un vector de dimensiune *n*: a = {a_1, a_2, …, a_n}, $a_i \in \{0,1\}$

```
is_a_solution(int a[], int k, int n){
    return (k == n); /* is k == n? */
}

construct_candidates(int a[], int k, int n, int c[], int *ncandidates){
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}

process_solution(int a[], int k){
    int i; /* counter */
    printf("{");
    for (i=1; i<=k; i++)
      if (a[i] == TRUE) printf(" %d",i); }
    printf(" }\n");
}
```

```
generate_subsets(int n){
    int a[NMAX]; /* solution vector */
    backtrack(a,0,n);
```

? In ce ordine se genereaza submultimile multimii {1, 2, 3} ?

Backtracking - Numararea tuturor cailor intre s si t in graf

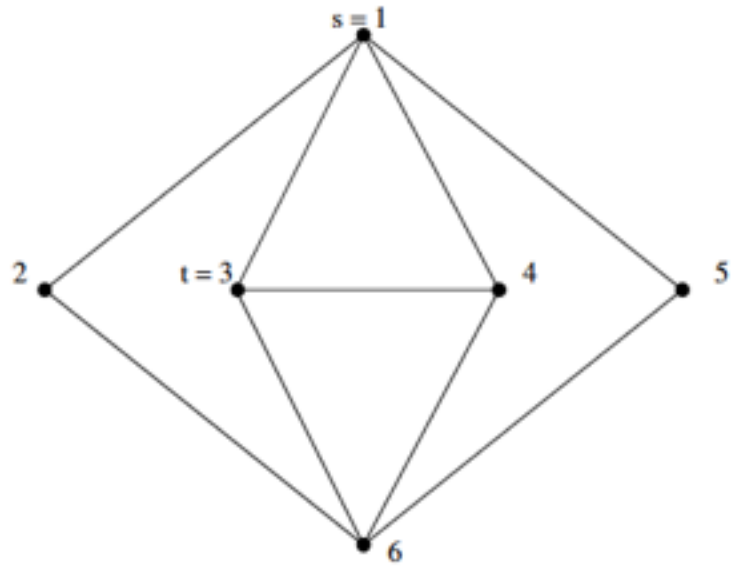
- lista candidat pentru prima pozitie: {s}; pentru a doua pozitie: {v | (s,v)∈E}; in general, la pasul *k* se adauga la lista de candidati varfurile adiacente lui a_k care nu apar in solutia partiala

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates) {
  int i; /* counters */
  bool in_sol[NMAX]; /* what's already in the solution? */
  edgenode *p; /* temporary pointer */
  int last; /* last vertex on current path */
  for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
  for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;
  if (k==1) { /* always start from vertex 1 */
     c[0] = 1;
     *ncandidates = 1;
  } else {
     *ncandidates = 0;
     last = a[k-1];
     p = g.edges[last];
     while (p != NULL) {
        if (!in_sol[ p->y ]) {
           c[*ncandidates] = p->y;
           *ncandidates = *ncandidates + 1;
        }
        p = p->next;
     }
  }
}
```

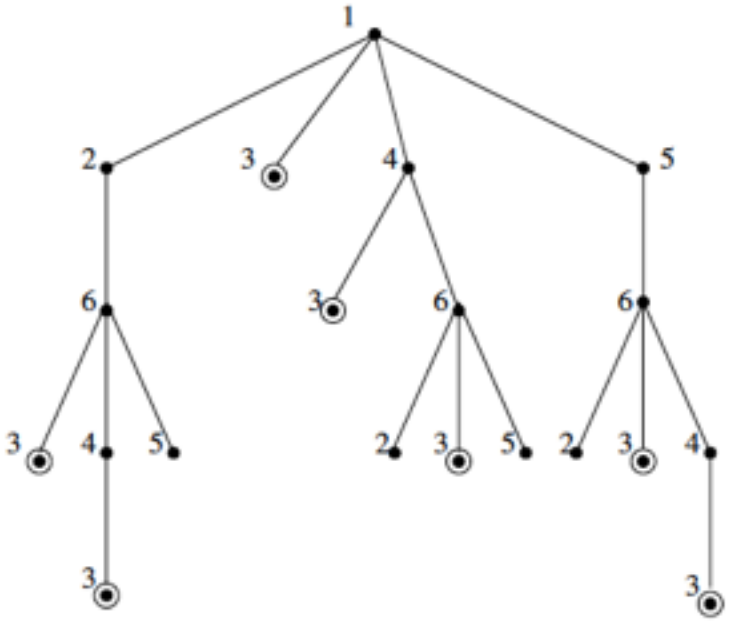
```
is_a_solution(int a[], int k, int t){
    return (a[k] == t);
}
process_solution(int a[], int k){
    solution_count ++; /* count all paths */
}
```

Backtracking - Numararea tuturor cailor intre s si t in graf

Graful, s si t



Arborele de cautare



Backtracking: SUDOKU

		1 2
	3 5	
6		7
7		3
	4	8
1		
	1 2	
8		4
5		6

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Candidatii pentru celula (i,j):

- intregii intre 1 si 9, care nu au aparut inca in randul i, coloana j si patratul dimensiune 3x3 care contine celula (i,j)
- revenirea se face de indata ce nu mai exista candidati viabili pentru o celula

Backtracking: SUDOKU

```
#define DIMENSION 9 /* 9*9 board */
#define NCELLS DIMENSION*DIMENSION /* 81 cells in a 9*9 problem */

typedef struct {
  int x, y;
} point;
/* x and y coordinates of point */

typedef struct {
  int m[DIMENSION+1][DIMENSION+1]; /* matrix of board contents */
  int freecount; /* how many open squares remain? */
  point move[NCELLS+1]; /* how did we fill the squares? */
} boardtype;

 is_a_solution(int a[], int k, boardtype *board) {
    if (board->freecount == 0)
      return (TRUE);
    else
      return(FALSE);
}
```

Backtracking: SUDOKU

```
construct_candidates(int a[], int k, boardtype *board, int c[],
int *ncandidates){
    int x,y; /* position of next move */
    int i; /* counter */
    bool possible[DIMENSION+1]; /* what is possible for the square */

    next_square(&x,&y,board); /* which square should we fill next? */
    board->move[k].x = x; /* store our choice of next position */
    board->move[k].y = y;
    *ncandidates = 0;
    if ((x<0) && (y<0)) return; /* error, no moves possible */
    possible_values(x,y,board,possible);
    for (i=0; i<=DIMENSION; i++)
        if (possible[i] == TRUE) {
          c[*ncandidates] = i;
          *ncandidates = *ncandidates + 1;
        }
}
```

Backtracking: SUDOKU

```
 make_move(int a[], int k, boardtype *board){
    fill_square(board->move[k].x,board->move[k].y,a[k],board);
}

unmake_move(int a[], int k, boardtype *board){
    free_square(board->move[k].x,board->move[k].y,board);
}

 process_solution(int a[], int k, boardtype *board){
    print_board(board);
    finished = TRUE;
}
```

Backtracking: SUDOKU

`next_square(&x,&y,board)`

- *selectie arbitrara* - primul/ultimul/aleator din celulele necompletate
- *selectia celei mai constranse celule* - celula cu cele mai putine valori candidat disponibile

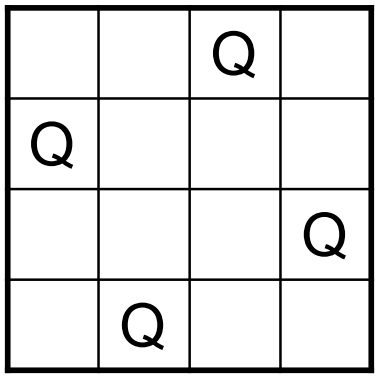
`possible_values(x,y,board,possible)`

- *numarare locala* - constrangerea le rand, coloana si sector
- *"look ahead"* - daca constrangerea provine de la o alta celula, astfel incat nu exista nici o valoare posibila pentru celula curenta

Pruning Condition		Puzzle Complexity		
`next_square`	`possible_values`	Easy	Medium	Hard
arbitrary	local count	1,904,832	863,305	never finished
arbitrary	look ahead	127	142	12,507,212
most constrained	local count	48	84	1,243,838
most constrained	look ahead	48	65	10,374

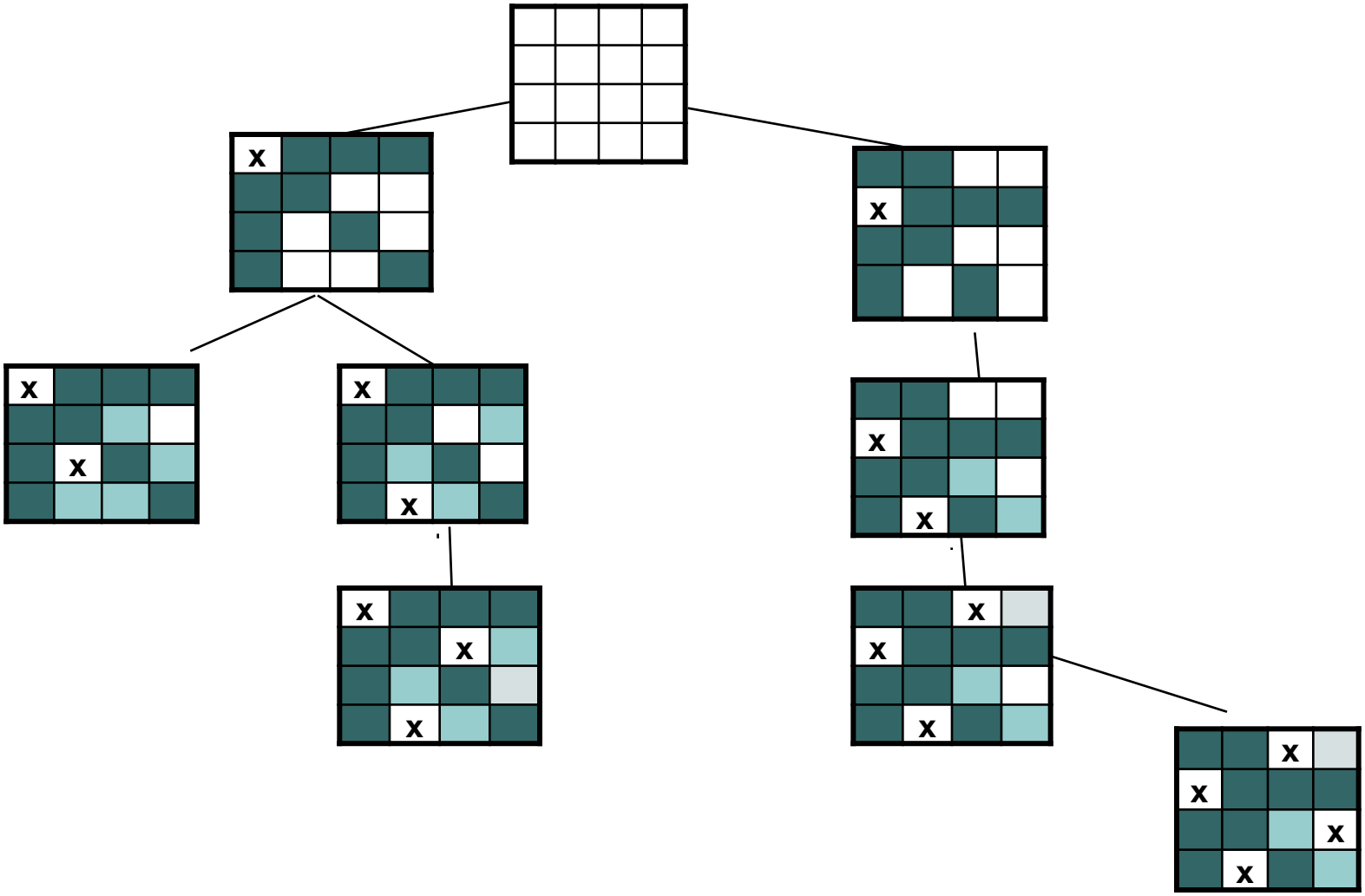
Backtracking: Problema celor n regine

- Cum asezam *n* regine pe o table de sah de dimensiune *nxn* astfel incat sa nu se atace, i.e. nu avem 2 regine pe aceeasi *linie*, *coloana* sau *diagonala*



- Strategie:
  - se incearca plasarea cate unei regine pe fiecare coloana
  - la fiecare coloana noua se plaseaza regina pe randul care nu este in conflict cu configuratia de pana atunci
  - daca nu este posibila plasarea reginei pe coloana curenta, se revine la coloana anterioara

Backtracking: Problema celor n regine



Backtracking: Problema celor n regine

```
nQueens(n)
{
   rnQueens(1, n)
}

positionOK(k)
{
  for i = 1 to k–1
      if (row[k] = row[i] ||
          abs(row[k]–row[i]) = k–i)
              return false;
   return true;
}
```

```
rnQueens(k,n) {
   for row[k]= 1 to n
     if (positionOK(k) )
       if ( k = n )
       {
          for i = 1 to n
              print(row[i] + " ");
          println;
       }
     else
        rnQueens(k+1, n)
}
```

Tehnica greedy: Inapoi la numararea restului...

- Un casier nu considera de fapt toate posibilitatile de numarare a sumei date
- In schimb, numara incepand de la cea mai mare valoare si trecand apoi la valori mai mici
- Exemplu: $\{d_1, d_2, \ldots, d_{10}\} = \{1, 1, 1, 1, 1, 5, 5, 10, 25, 25\}$. Suma 32: 25, 5, 1, 1
- Odata ce o moneda a fost selectata, nu mai sunt considerate solutii fara acea moneda
- Daca bancnotele/monezile sunt gata sortate - timpul de rulare a acestui algoritm este O(n).
- Nu garanteaza gasirea solutiei optime!
  - E.g. {1, 1, 1, 1, 1, 10, 10, 15}, suma 20

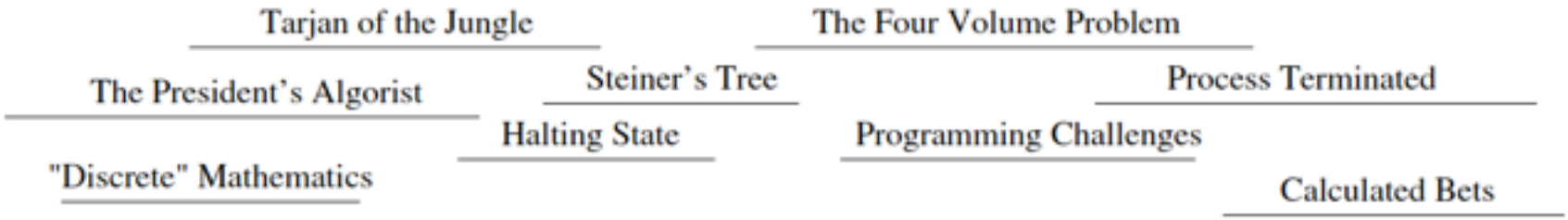
Tehnica greedy

- Un algoritm **greedy** selecteaza mereu alternativa cea mai buna la acel moment, in speranta ca aceea va duce la solutia optima globala.
- Nu garanteaza gasirea solutiei optime, dar exista situatii in care acest lucru este posibil:
  - problema selectiei activitatilor
  - arborele minim de acoperire (Prim, Kruskall)
  - gasirea cailor minime in graf (Dijkstra)
  - etc.

Tehnica greedy: Selectia activitatilor

- Avem o multime $S=\{a_1, a_2, \ldots a_n\}$ de n activitati care utilizeaza o *resursa* (e.g. sala de concerte)
- Fiecare activitate are timp de *inceput*, s_i, si timp de *sfarsit*, f_i, cu $0 \leq s_i < f_i < \infty$
- Daca este selectata, activitatea a_i ocupa intervalul $[s_i, f_i)$
- a_i si a_j sunt *compatibile* daca $[s_i, f_i) \cap [s_j, f_j) = \phi$
- Se cere sa se selecteze ***sub-multimea maxima de activitati mutual compatibile***.

Tehnica greedy: Selectia activitatilor



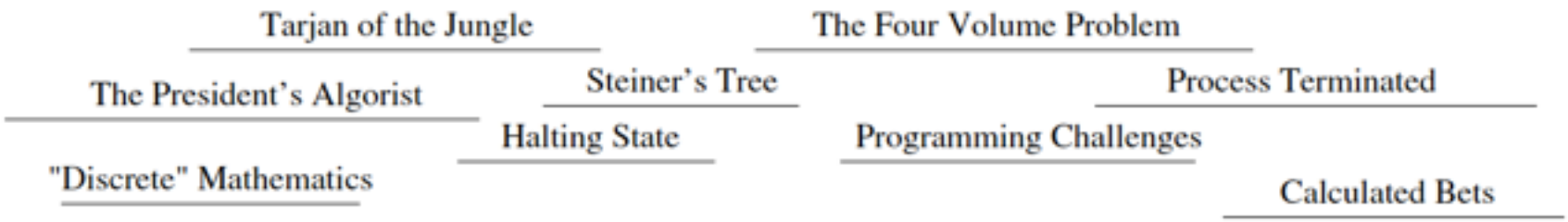
- ***In ce ordine selectam activitatile astfel incat sa avem garantia solutiei optime?***

Tehnica greedy: Selectia activitatilor

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

- Presupunem ca activitatile sunt ordonate crescator dupa timpul de finalizare:
  - $0 \leq f_1 \leq f_2 \leq f_3 \ldots \leq f_{n-1} \leq f_n$

Tehnica greedy: Selectia activitatilor



- Ce solutie obtinem pentru aceasta instanta de problema?
  - A={*"Discrete" Mathematics*, *Halting State, Programming Challenges, Calculated Bets*}

Tehnica greedy: Problema rucsacului

- Se da:
  - O multime de *n* obiecte, fiecare avand o anumita *greutate* si o anumita *valoare (profit)*
  - Un rucsac de o anumita capacitate (poate contine o anumita greutate)
- Se cere: sa se selecteze sub-multimea de obiecte care incape in rucsac si maximizeaza *valoarea* totala
- Modelare:
  - w_i *greutatea* obiectului ***i***,
  - p_i *profitul* elementului ***i***
  - ***W*** *capacitatea* rucsacului
  - x_i variabila din vectorul solutie, care ia valoarea 1 cand obiectul ***i*** este carat in rucsac, si 0 altfel

Tehnica greedy: Problema rucsacului

- Dandu-se multimile $\{w_1, w_2, \ldots, w_n\}$ si $\{p_1, p_2, \ldots, p_n\}$, obiectivul este maximizarea:

$$\sum_{i=1}^{n} p_i x_i$$

- supusa constrangerii: $\sum_{i=1}^{n} w_i x_i \leq W$

- Seamana aceasta problema cu problema *numararii restului*?
- O solutie posibila - bazata pe *backtracking (exercitiu)*

Tehnica greedy: Problema rucsacului

- Euristici posibile pentru tehnica greedy:
  - ***Greedy dupa profit***
    - la fiecare pas se selecteaza obiectul cu profit maxim
    - selecteaza cele mai profitabile obiecte intai
  - ***Greedy dupa greutate***
    - la fiecare pas se selecteaza obiectul de greutate minima
    - incearca sa maximizeze profitul maximizand numarul de obiecte din rucsac
  - ***Greedy dupa densitatea de profit***
    - la fiecare pas se selecteaza obiectul care area cea mai mare densitate de profit p_i/w_i
    - incearca sa maximizeze profitul prin selectia obiectelor cu cel mai mare profit per unitate de greutate

Tehnica greedy: Problema rucsacului

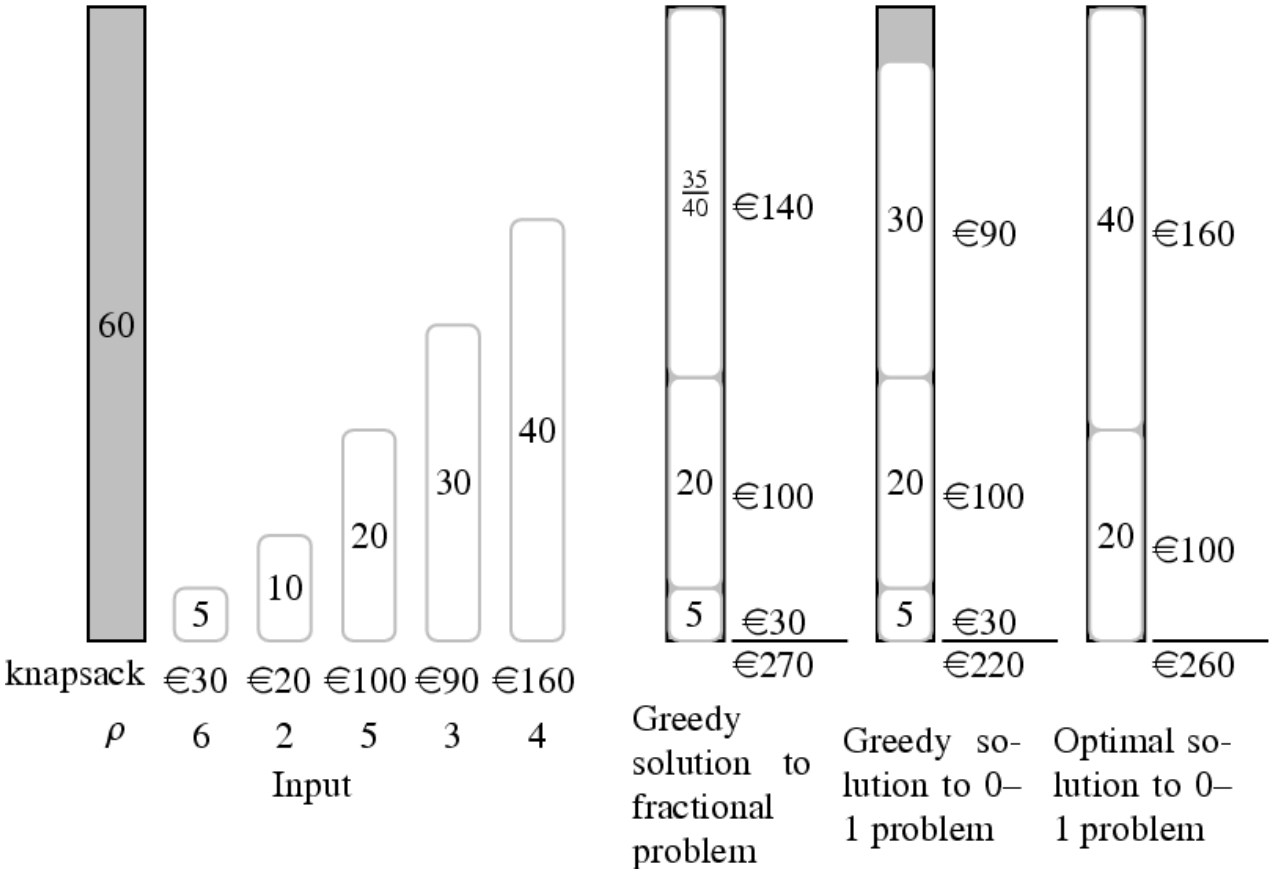
- $W = 100$

				greedy by			
i	w_i	p_i	p_i/w_i	profit	weight	density	optimal solution
1	100	40	0.4	yes	no	no	no
2	50	35	0.7	no	no	yes	yes
3	45	18	0.4	no	yes	no	yes
4	20	4	0.2	no	yes	yes	no
5	10	10	1.0	no	yes	yes	no
6	5	2	0.4	no	yes	yes	yes
total weight				100	80	85	100
total profit				40	34	51	55

- Nu garanteaza gasirea solutiei optime!

Tehnica greedy: Problema rucsacului - varianta fractionara

- Obiectele pot fi *selectate partial* - i.e. putem selecta o *fractiune* din obiect - la o fractiune de profit si greutate



Tehnica greedy: Codurile Huffman

- Tehnica de compresie a datelor
- Date: secventa de caractere
- Se utilizeaza un ***tabel*** care stocheaza frecventa de aparitie a fiecarui caracter
- Pe baza valorilor din tabel se construieste o reprezentare *binara optima*, *de lungime variabila*
- *Exemplu*:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

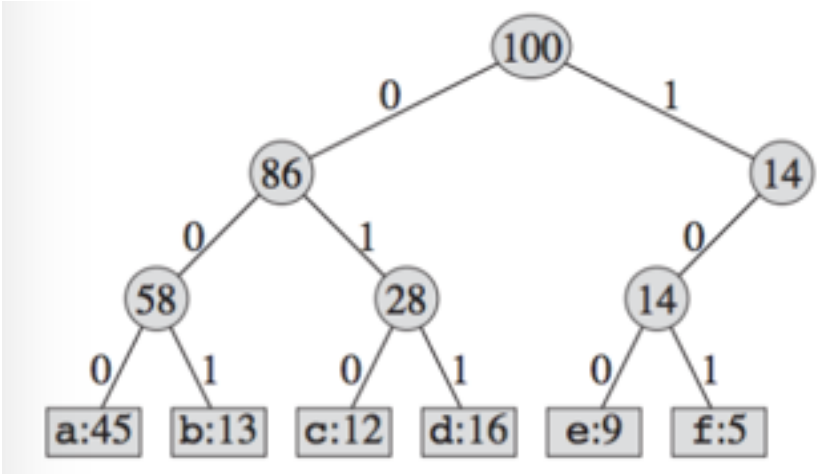
Tehnica greedy: Codurile Huffman

- Fiecarei litere ii asociem un ***cod binar***:
  - de *lungime fixa*: 300.000 bits
  - de *lungime variabila*: 224.000 bits (you do the math)
- **Coduri prefix**:
  - consideram doar codurile care nu apar ca si prefix al altui cod
    - metoda optima de compresie
  - simplifica decodificarea: codul care incepe fisierul codat nu este ambiguu; il identificam, il traducem, si continuam pe restul fisierului

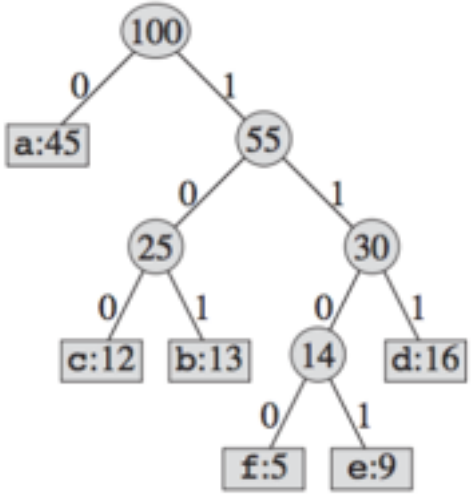
Tehnica greedy: Codurile Huffman

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- 001011101 = 0.0.101.1101 = aabe
- Reprezentare: **arbore binar**; codul = calea de la radacina la o frunza; 0 - “go left”, 1 - “go right”



cod de lungime fixa



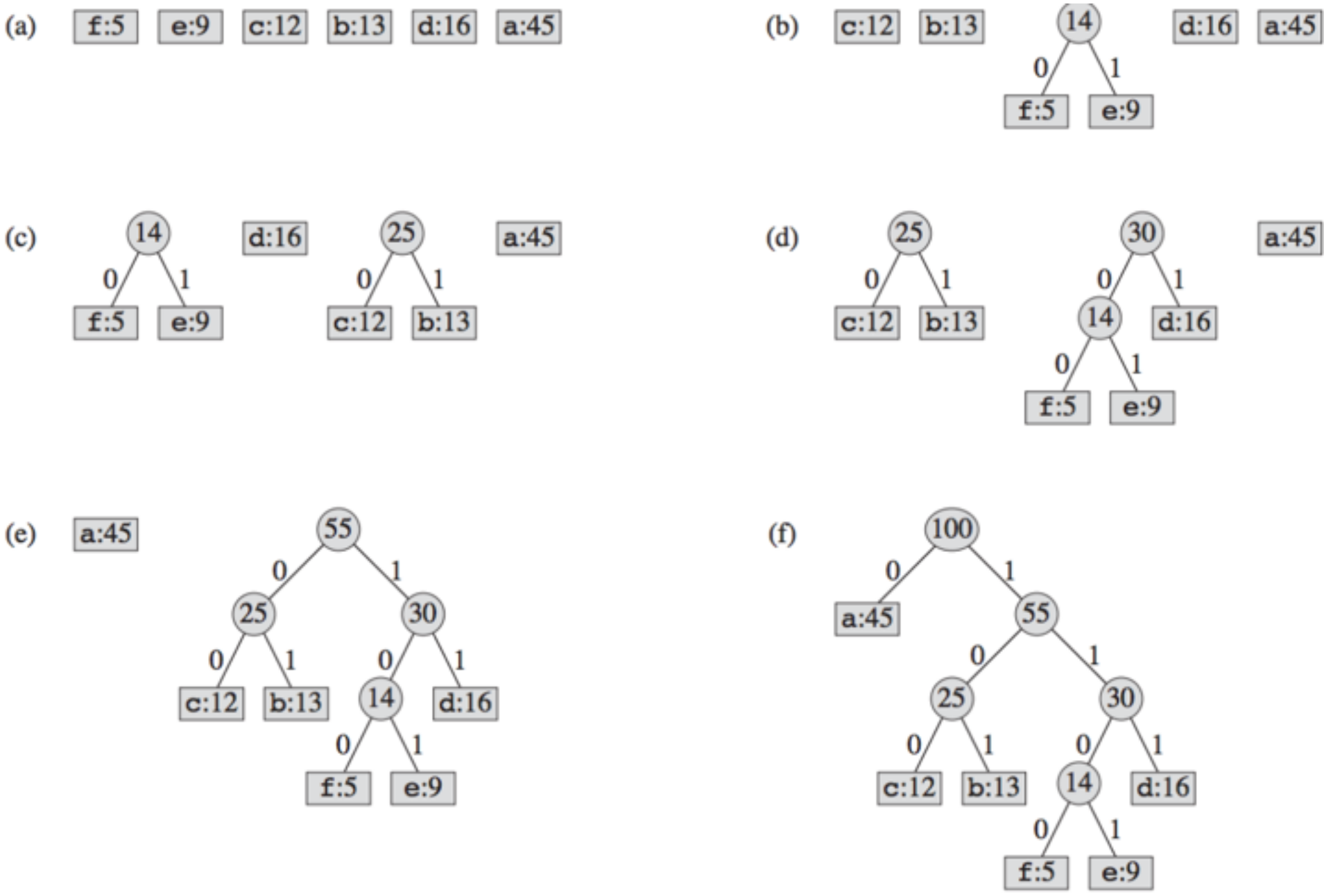
cod de lungime variabila

Tehnica greedy: Codurile Huffman

- Cum construim codificarea?
  - C - multimea de *n* caractere
  - strategie bottom-up
  - C*oada de prioritati* (minim) - Q, construita pe frecventa, pentru a identifica cele 2 obiecte de frecv. minima pentru a fi unite

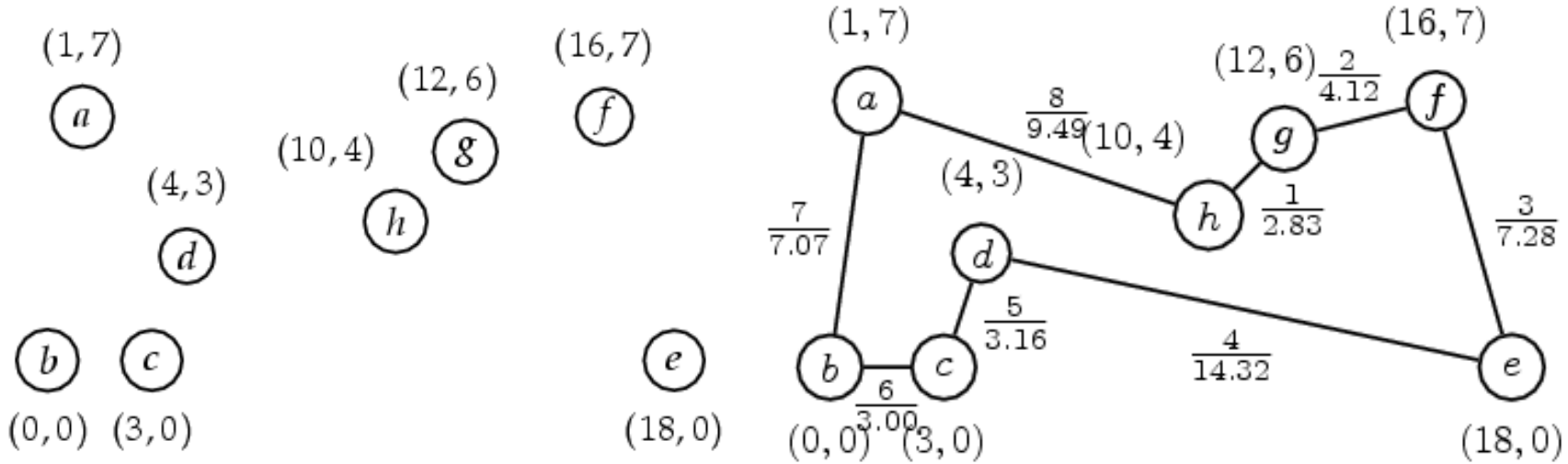
```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Eficienta?


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Tehnica greedy: TSP

- Ciclu Hamiltonian
  - Fiind dat un graf, un ciclu (tur) Hamiltonian este un ciclu simplu care trece prin toate varfurile din graf
- TSP
  - Fiind dat un graf cu costuri pe muchii, gasiti turul simplu de cost minim.



Tehnica greedy: TSP

- Functioneaza pe grafuri complete doar
- Se sorteaza muchiile in ordinea crescatoare a ponderilor
- Pornind de la muchia de cost minim, se selecteaza pe rand cate o muchie noua care lungeste drumul curent, astfel incat:
  - muchia noua sa nu genereze grad >= 3 la nici un nod
  - muchia nu formeaza un ciclu, decat daca am ajuns la ultimul nod din graf

Bibliografie

- S. Skiena: The Algorithm Design Manual, cap 7.
- Th. Cormen et al.: Introduction to Algorithms, cap. 16