Algoritmid ja andmestruktuurid

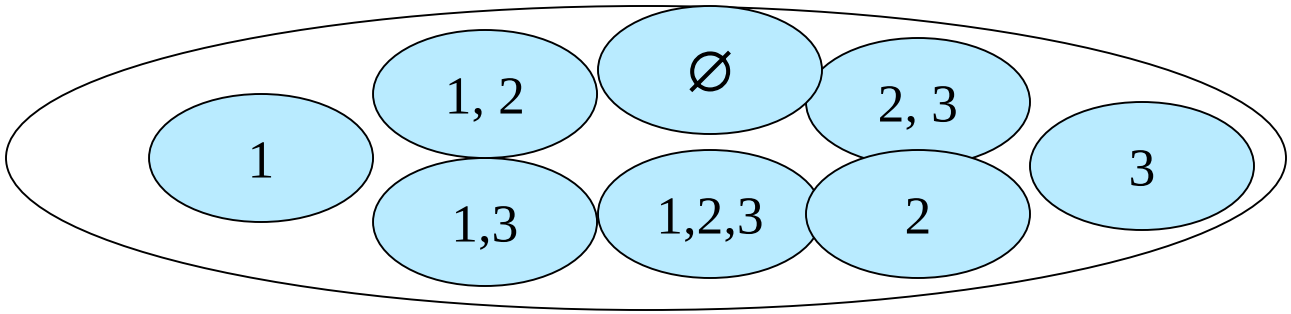
- Kombinatoorsed optimiseerimisülesanded
- Tagasivõtmisega (*backtracking*) algoritmid
- Hargne ja kärbi (*branch and bound*) algoritmid

Kombinatoorsed optimiseerimisülesanded

- Kasutatakse ülesannete lahendamiseks, kus mingist hulgast tuleb välja otsida hulk mingile kriteeriumile vastavaid elemente.
  - parim tee, parim paigutus
- Eesmärgiks võib olla leida
  - üks lahendus
  - kõik lahendused
  - optimaalne lahendus
- Käsitletakse erinevates valdkondades
  - algoritmiteooria, keerukusanalüüs
  - Operatsioonianalüüs
  - tehisintellekt

Lahenduse otsimine

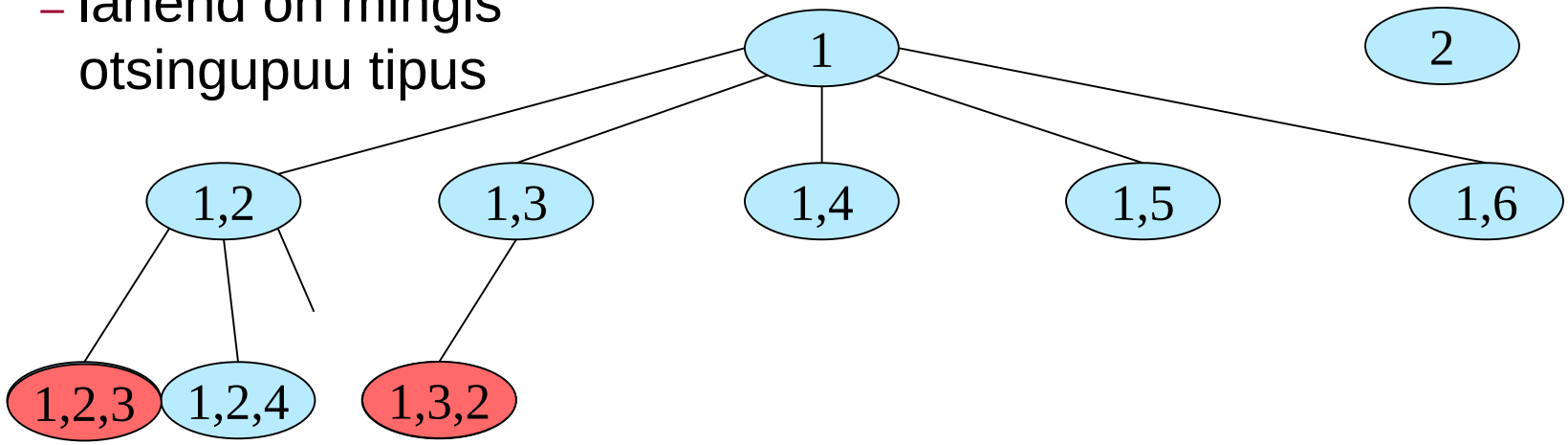
- Lahenduse annab mingi parameetrite (sisendite) kombinatsioon - konfiguratsioon
  - punktide läbimise järjekord
  - elementide valik mingist hulgast
  - mingi paigutus võimalike paigutuste hulgast
  - jne
- Igal hulgal suurusega n on 2^n alamhulka, $n!$ permutatsiooni
  - sellest tuleneb eksponentsiaalne või suurem keerukus



1,2,3
1,3,2
2,1,3
2,3,1
3,1,2
3,2,1

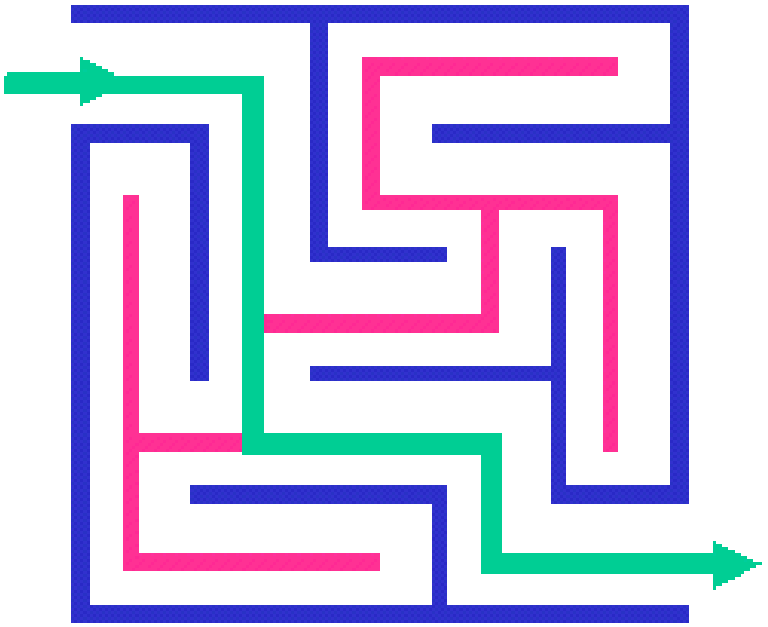
Lahenduse otsimine

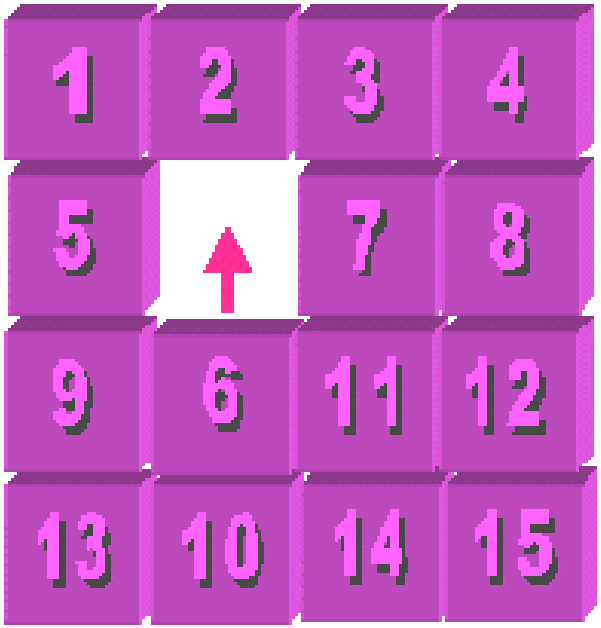
- Tavaliselt on võimalik organiseerida parameetrite kombinatsioonid (konfiguratsioonid) mingisse puukujulisse struktuuri
  - tavaliselt ei koostata sellist andmestruktuuri, puu on lihtsalt abstraktsioon - tegevuste organiseerimise struktuur
- Lahendit on võimalik otsida puu läbimisega
  - lahend on mingis otsingupuu tipus



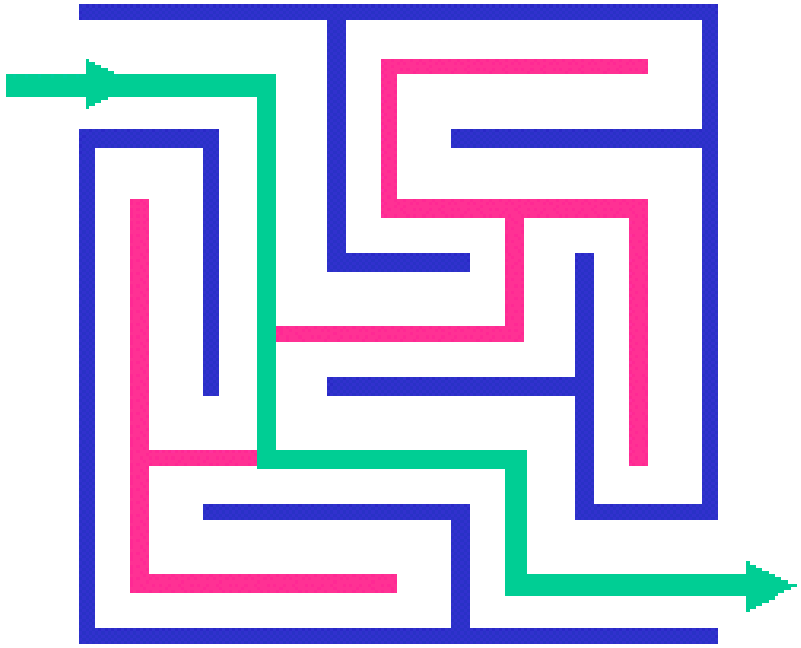
Labürint, 15 mäng

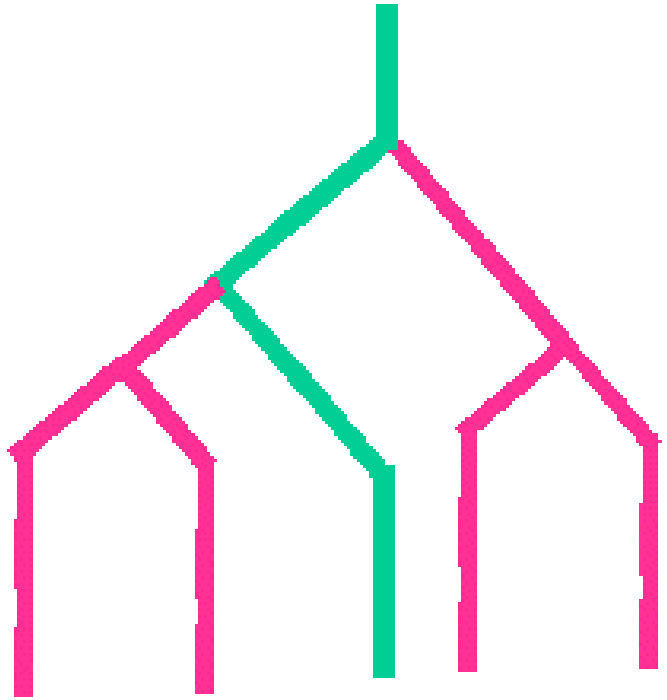
- Meil on igal ajahetkel mitu võimalikku valikut
- Kui läksime ummikteed, siis pöördume eelneva seisu juurde tagasi



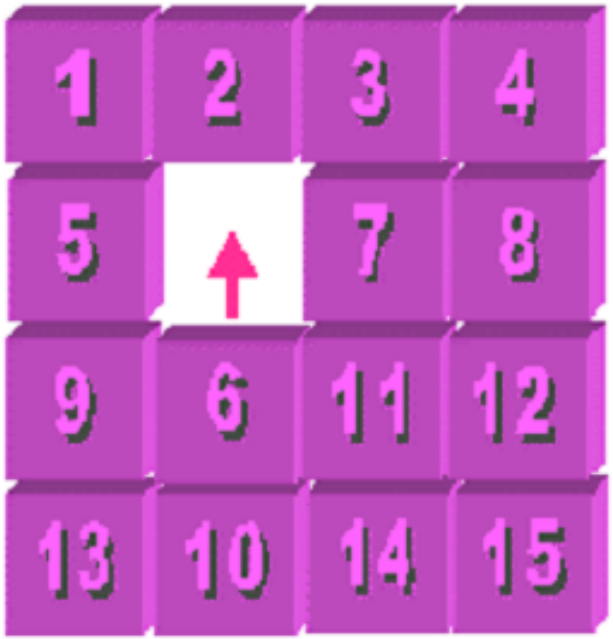


Otsingupuu

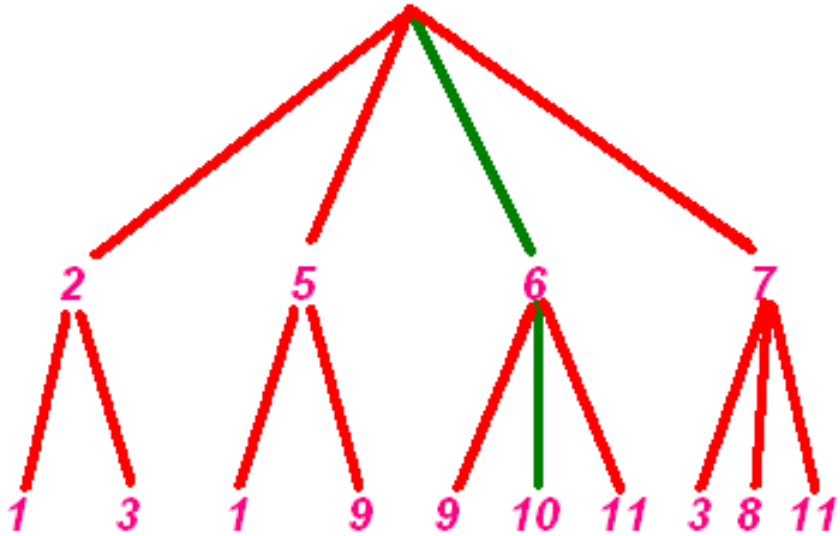




Otsingupuu



6
10
14
15



Markide paigutamise probleem

- Meil on kasutada 10, 20 ja 50 sendised margid. Leida millises kombinatsioonis tuleks marke panna mingi summa maksmiseks, et markide arv oleks minimaalne. Millise algoritmiga saab sobilikku markide komplekti valida?
- Aga kui on kasutada 10, 40 ja 50 sendised margid?
- Algoritm stiilis “võta suurimaid münte niipalju kui saad ja siis sellele järgneva suurusega jne” (ahne algoritm) ei tööta! Tõesta!

2.70 minimaase arvu markidega

50 sendiste markide arv	40 sendiste markide arv	10 sendiste markide arv	vajalike markide arv kokku
5	0	2	7
4	1	3	8
3	3	0	6

Markide paigutamine võrrandina

- Sisuliselt otsime lahendust võrrandile

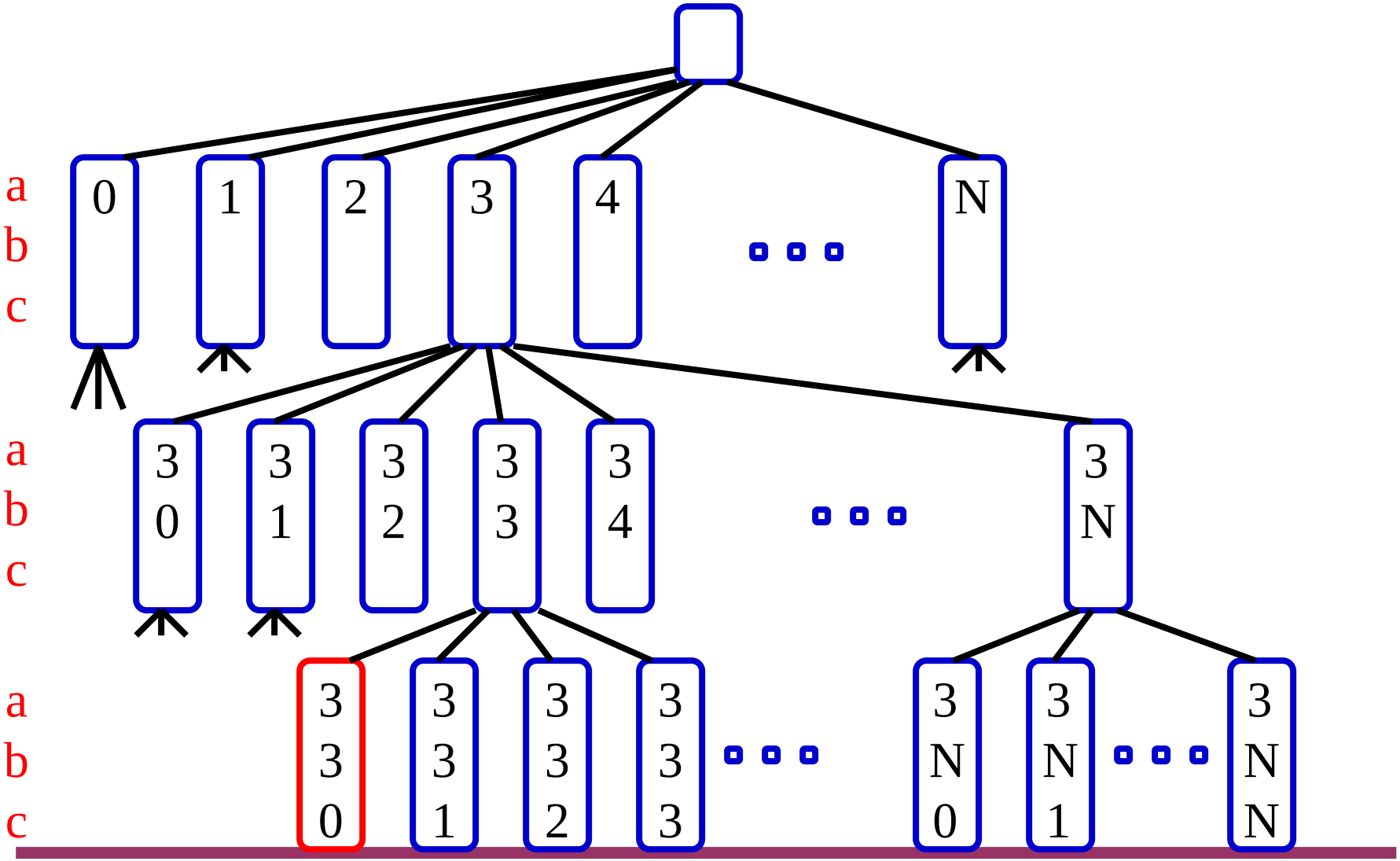
- a*50 + b*40 + c*10 = summa
  nii et a+b+c oleks minimaalne

Täielik algoritm (jõumeetod, *brute force)*

```
findStamps(N):
  min = N+1
  for a in 0 … N:
    for b in 0 … N:
      for c in 0 … N:
        if a*50 + b*40 + c*10 == n:
          if a+b+c < min
            min = a+b+c
            save <a,b,c>
print saved <a, b, c>
```

Keerukus o(n^3)

Markide paigutamise ülesande otsingupuu

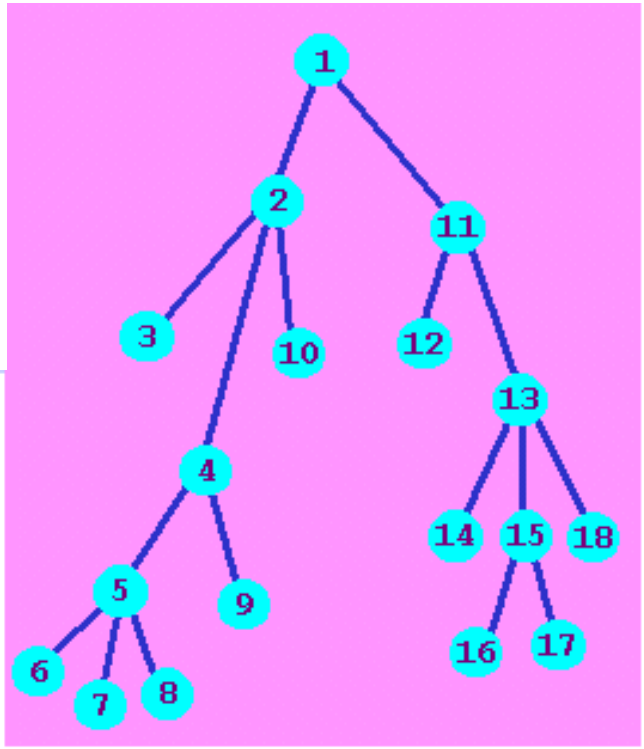


Täielik (*brute-force*) tagasivõtmisega (*backtracking*) algoritm

- Täielik algoritm vaatab läbi kõik parameetrite kombinatsioonid
  - üldine meetod täieliku otsingu teostamiseks
  - optimiseerimisülesandel väljastab parima konfiguratsiooni
  - valikuülesandel väljastab sobiva(d) konfiguratsiooni(d)
- Tagasivõtmine (*backtracking*)
  - konfiguratsioon koostatakse osade kaupa - liigutakse mööda mingit puu haru alla
  - kui konfiguratsiooni enam elemente lisada ei saa, siis “tagurdatakse” ja valitakse alternatiivne element - liikumine puus üles ja kõrvaloleva naabrini
- Sisuliselt sügavuti otsing otsingupuus

Sügavuti otsimine

```
void depth_first_search(node v)
{ node u;
  visit v;
  for(each child u of v)
    depth_first_search(u);
}
```



Lootustandvad järglased (*promising nodes*)

- Tihti saab mingis otsingupuu harus olles otsustada, kas see haru on
  - lootusetu – selle kaudu pole võimalik lahenduseni jõuda
  - lootustandev (*promising*) – on võimalik, et lahendus on selles harus
- Otsida on mõtet ainult lootustandvatest harudest
- Kärpimine – loobutakse lootusetu haru läbiotsimisest

node - (graafi/puu) tipp, sõlm, haru

Üldine tagasivõtmisega rekursiivne algoritm

```
void checknode(node v)
{
  node u;

  if(promising(v))
    if(there is a solution at v)
      write the solution;
    else
      for(each child u of v)
        checknode(u);
}
```

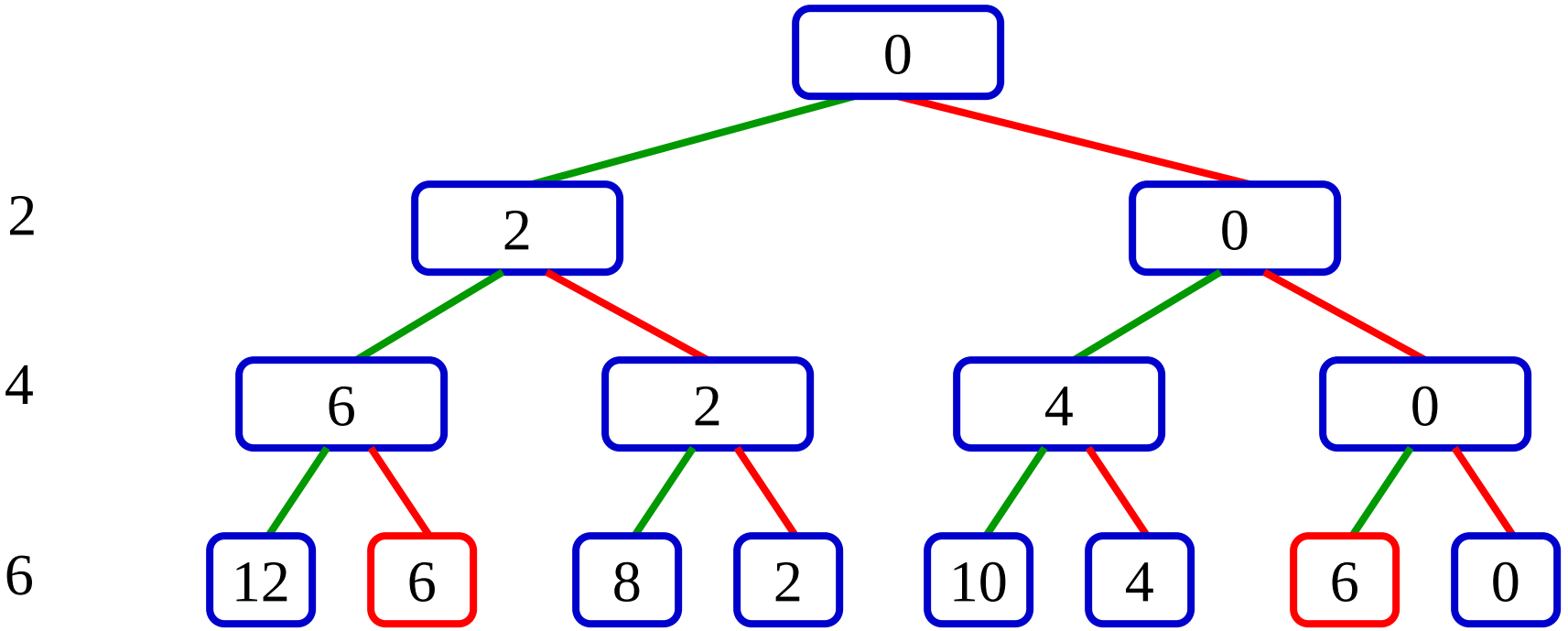
Alamhulkade summa

- On antud n täisarvu ja summa W.
- Leida kõik kombinatsioonid antud arvudest, mis annaksid summerides W
- Näiteks:

{5, 6, 10, 11, 16}, W=21

Lahendused: {5, 6, 10}, {5, 16}, {10, 11}

Alamhulkade summa otsingupuu



{2, 4, 6} W = 6

Otsingupuu kärpimine

- Lisatavad ühikud on sorteeritud suurenevas järjekorras
- Lootusetud harud kärbitakse, lahendust otsitakse lootustandvatest (*promising*) harudest
- Haru on lootusetu kui
  - järgneva ühiku lisamisega minnakse üle lõppsumma

    $kaal + w_{i+1} > W$

    lisatavad elemendid on suurenevas järjekorras!
  - kõigi lisamata ühikute summa on väiksem kui puuduolev summa

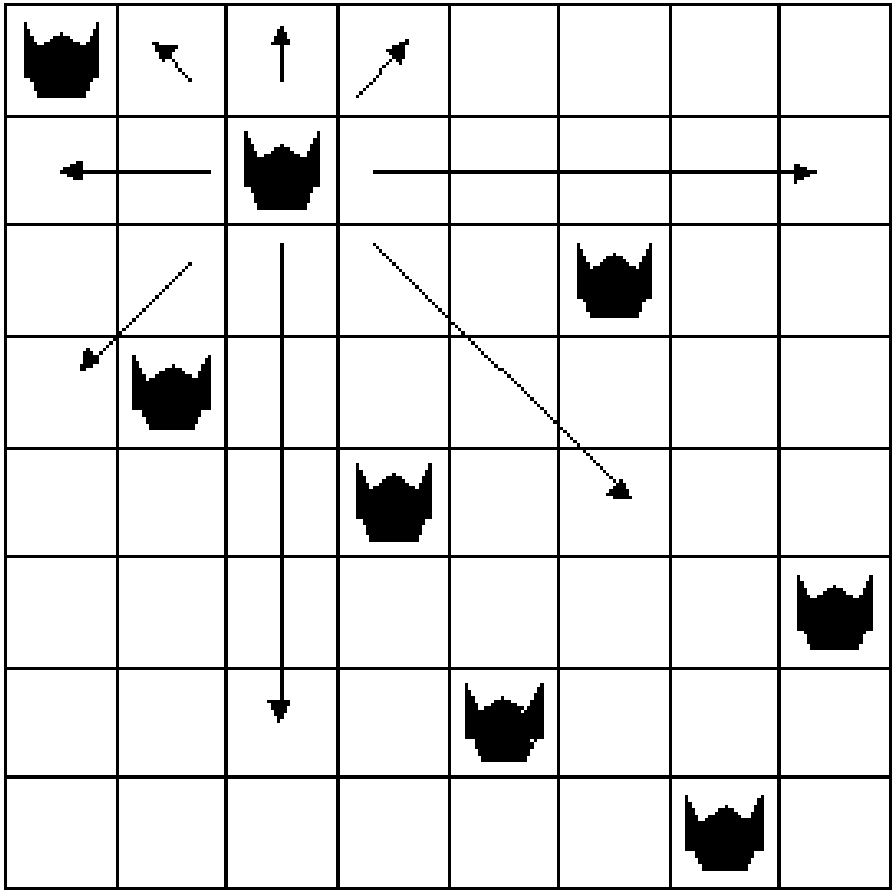
    $kaal + w\_total < W$

Alamhulkade summa algoritm

```
void sum_of_subsets (index i, int weight, int total)
{ if(promising(i))
    if(weight == W)
      print(include[1] .. include[i]);
    else {
      include[i+1] = “yes”;
      sum_of_subsets(i+1, weight+w[i+1], total-w[i+1]);
      include[i+1] = “no”;
      sum_of_subsets(i+1, weight, total-w[i+1]);
    }
}
bool promising (index i) {
  return(weight + total >= W) &&
        (weight == W || weight + w[i+1] <= W);
}
sum_of_subsets(0,0,sum(w[i]))    // alguse väljakutse
```

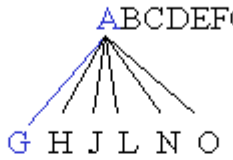
Lippude paigutamise probleem

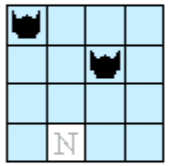
Paigutada NxN malelauale n lippu, nii et ükski neist ei oleks teisega samal real, veerul ega diagonaalil.

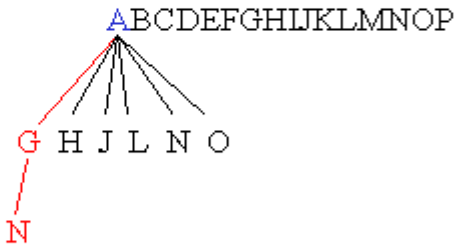


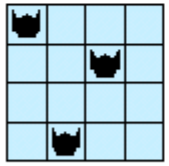
Lahendus jõumeetodil

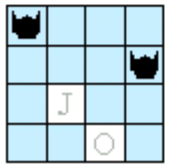
ABCDEFGHIJKLMNOP

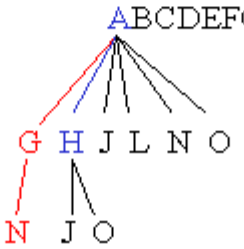


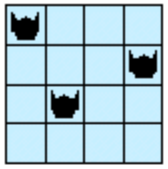


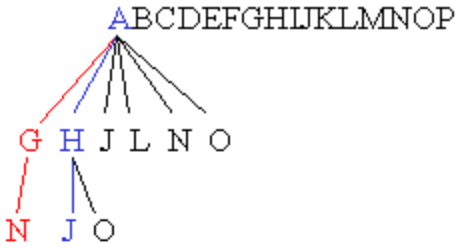


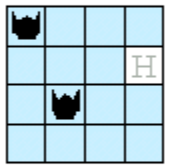


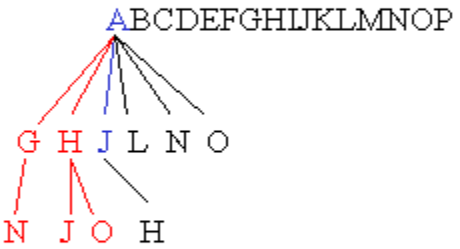


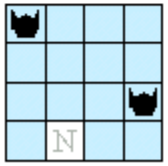


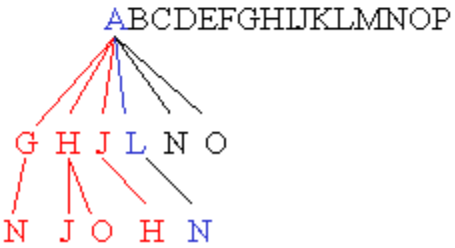




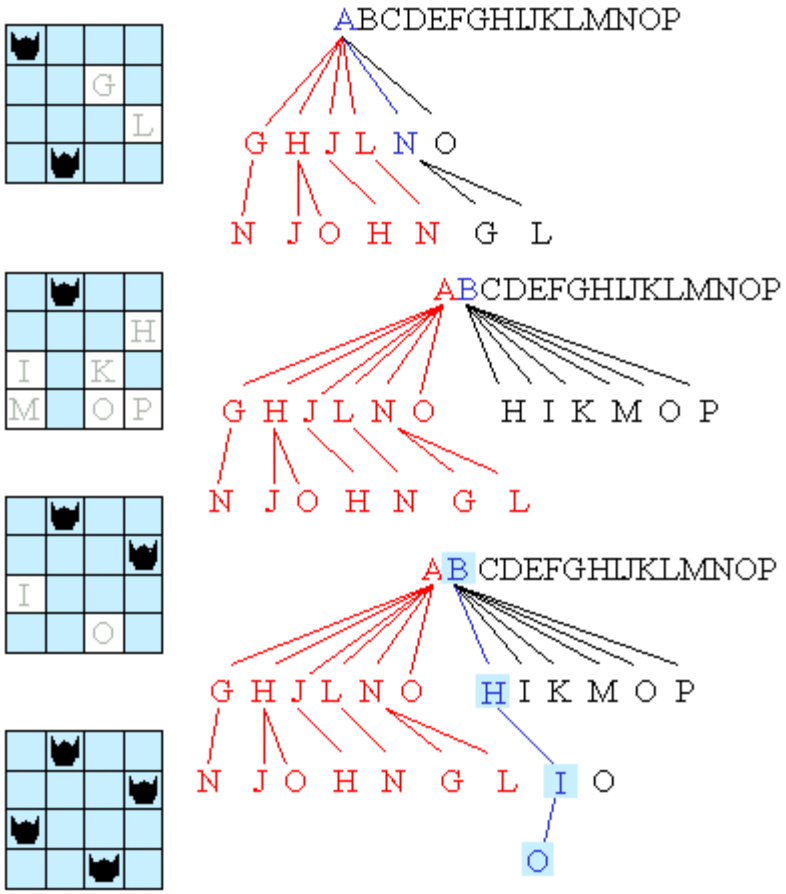








Lahendus jõumeetodil



Parem lahendus

- Paneme tähele et
  - on N rida ja N veergu ja N lippu
  - igas reas (ja veerus) saab olla ainult üks lipp
  - igas reas peab olema üks lipp
  - meie praegune algoritm kontrollib ka seise kus samas reas on mitu lippu

- Kodeerime paremini
  - teame, et igas reas saab ja peab olema üks lipp
  - tähistame $V[i]$-ga veeru, kus asub i-nda rea lipp
  - st kodeerime osa kitsendusi algoritmi, et otsingupuu tuleks väiksem

Otsingupuu

```
V[1] = 2
V[2] = 4
V[3] = 1
V[4] = 3
```

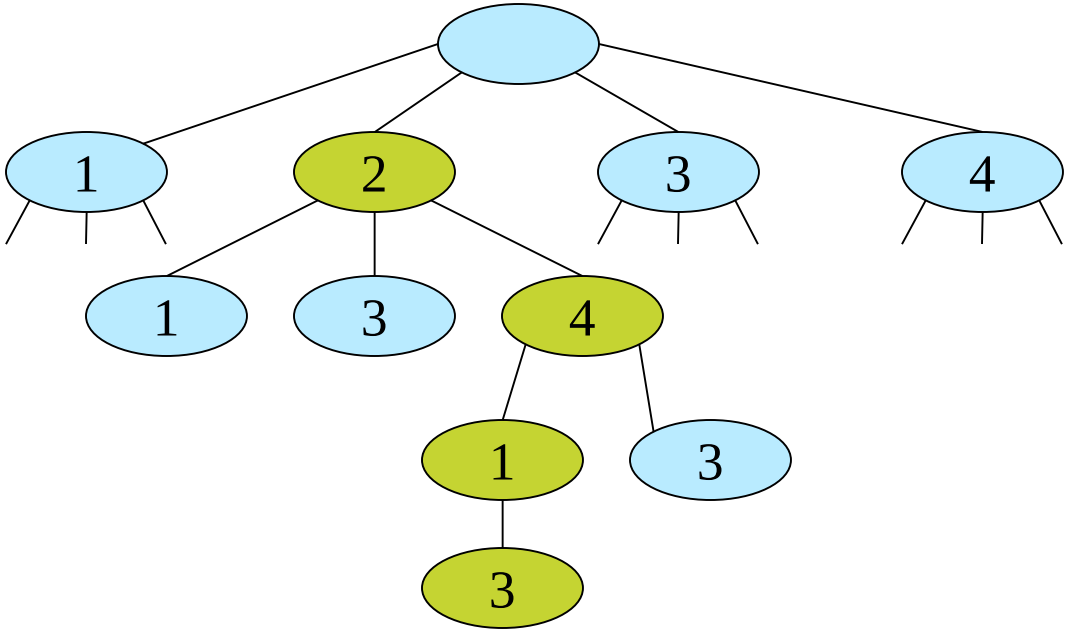
- Otsingupuusse jääb nii n! (4!=24) tippu, mitte enam $(n^2)^n$ (16^4 = 65534).
- Niipalju kui võimalik tuleks lahenduse kitsendusi sisse kodeerida algoritmi!

1. rea lipp

2. rea lipp

3. rea lipp

4. rea lipp



Lippude paigutamise algoritm

```
void queens (index i)
{
  index j;

  if(promising(i))
    if(i==n)
      print(V[i] ... V[n])
    else
      for(j=1; j<=n; j++){
        V[i+1]=j;
        queens(i+1);
      }
}
```

```
bool promising(index i)
{
  index k;
  bool switch;
  k = 1;
  switch = true;
  while(k < i && switch){
    if(V[i] == V[k] ||
       abs(V[i]-V[k])==i-k)
      switch = false;
    k++;
  }
  return switch;
}
```

Algoritmide võrdlus

N	Kontrollitud lahendusi algoritm 1 (n^2 järglast)	Kontrollitud lahendusi algoritm 2 (i - 1 järglast)	Kontrollitud sõlmi tagasi-võtmisega	Lootust-andvaid sõlmi tagasi-võtmisega
8	19173961	40320	15761	2057
14	$1.2*10^{16}$	$8.72*10^{10}$	$3.78*10^{8}$	$2.74*10^{7}$

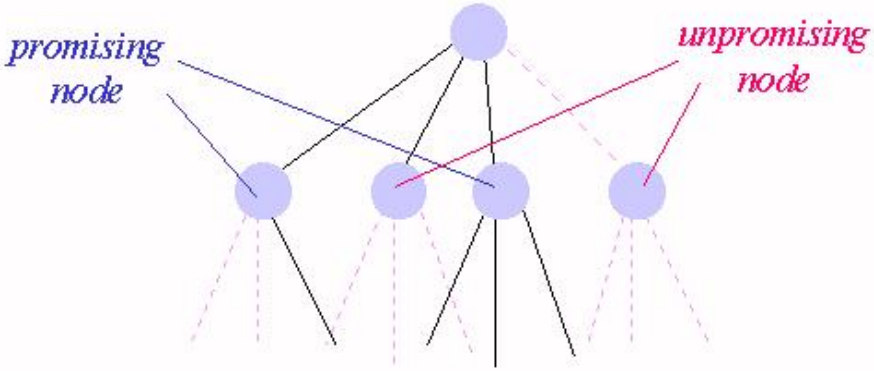
Hargne ja kärbi (*Branch and Bound*) strateegia

- Universaalne meetod diskreetsete optimiseerimisülesannete lahendamiseks
  - On olemas konfiguratsioonide hulk S(*k*)
  - On olemas sihifunktsioon f(*k*)
  - Tulemuseks konfiguratsioon k_{min}, mis minimiseerib f(*k*)
- Sarnaneb tagasivõtmisega (*backtracking*) strateegiale:
  - lahendust otsitakse konfiguratsioonide puust
  - otsingujärjekord võib erineda sügavuti otsingust
  - oluline on sihifunktsiooni olemasolu

Hargne ja kärbi (*Branch and Bound*) strateegia

Algoritmi olek hõlmab konfiguratsioonide alamhulka $S_i(k)$

- **Hargne** - alahulke jagatakse väiksemateks alamhulkadeks $S_j(k)$
- **Tõke** - igale uuele alamhulgale arvutatakse hinnang (*bound*), ehk **alumine tõke**. Ükski selle alamhulga konfiguratsioon ei anna sellest väiksemat sihifunktsiooni väärtust.
- **Kärbi** - Kui alumine tõke on suurem kui teadaolev parim lahendus, ehk **ülemine tõke**, siis kärbitakse see alamhulk.



Hargne ja Kärbi tagasivõtmisega

- **Sarnane tavalisele tagasivõtmisega algoritmile**
- **Meeles peetakse parimat leitud tulemust**

```
void checknode(node v)
{
  node u;

  if(value(v) is better than best)
    best = value(v);
  if(promising(v))
    for(each child u of v)
      checknode(u);
}
```

Üldine Hargne ja Kärbi meetod

Leia esialgne heuristiline lahend (mitteoptimaalne), et saada esialbne parim lahend (ülemine tõke)

Loo järjekord Q konfiguratsioonihulkad $S_i(k)$ hoidmiseks

while järjekord pole tühi

võta üks konfiguratsioonihulk järjekorrast

if see sisaldab ainult üht konfiguratsiooni ja see on parema sihifunktsiooniga parim lahend, siis on see parim

Jaga $S_i(k)$ alamhulkadeks $S_j(k)$

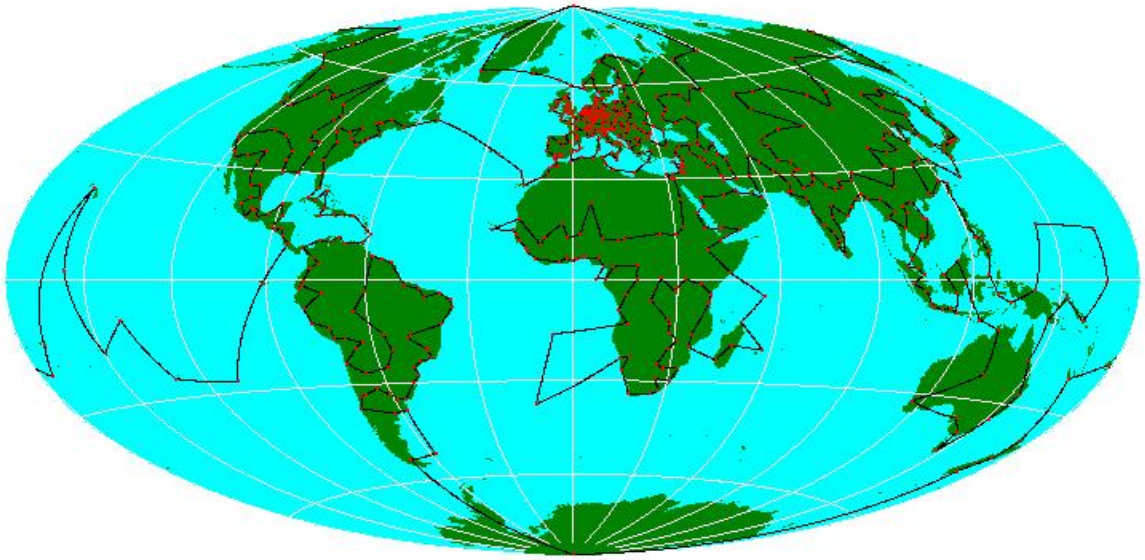
iga alamhulga $S_j(k)$ kohta arvuta hinnang (alumine tõke)

if hinnang on väiksem kui ülemine tõke

lisa $S_j(k)$ järjekorda

Rändkaupmehe (TSP) ülesanne

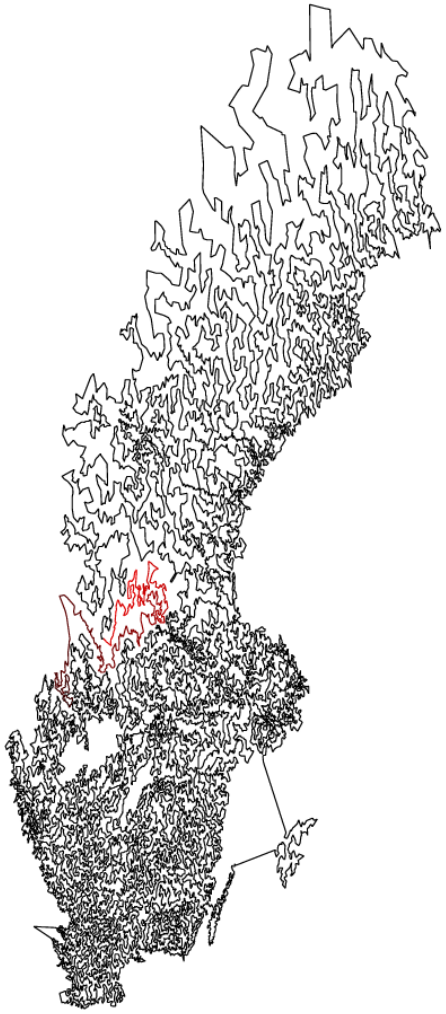
- Leida etteantud graafist lühim tee mis:
  - Algab ja lõpeb samas tipus (kohas)
  - Läbib igat tippu (kohta) täpselt ühe korra
- ATSP – asümmeetriline ülesanne, teepikkused AB ja BA ei pea olema võrdsed
- Eukleidiline TSP
  $AB + BC \geq AC$



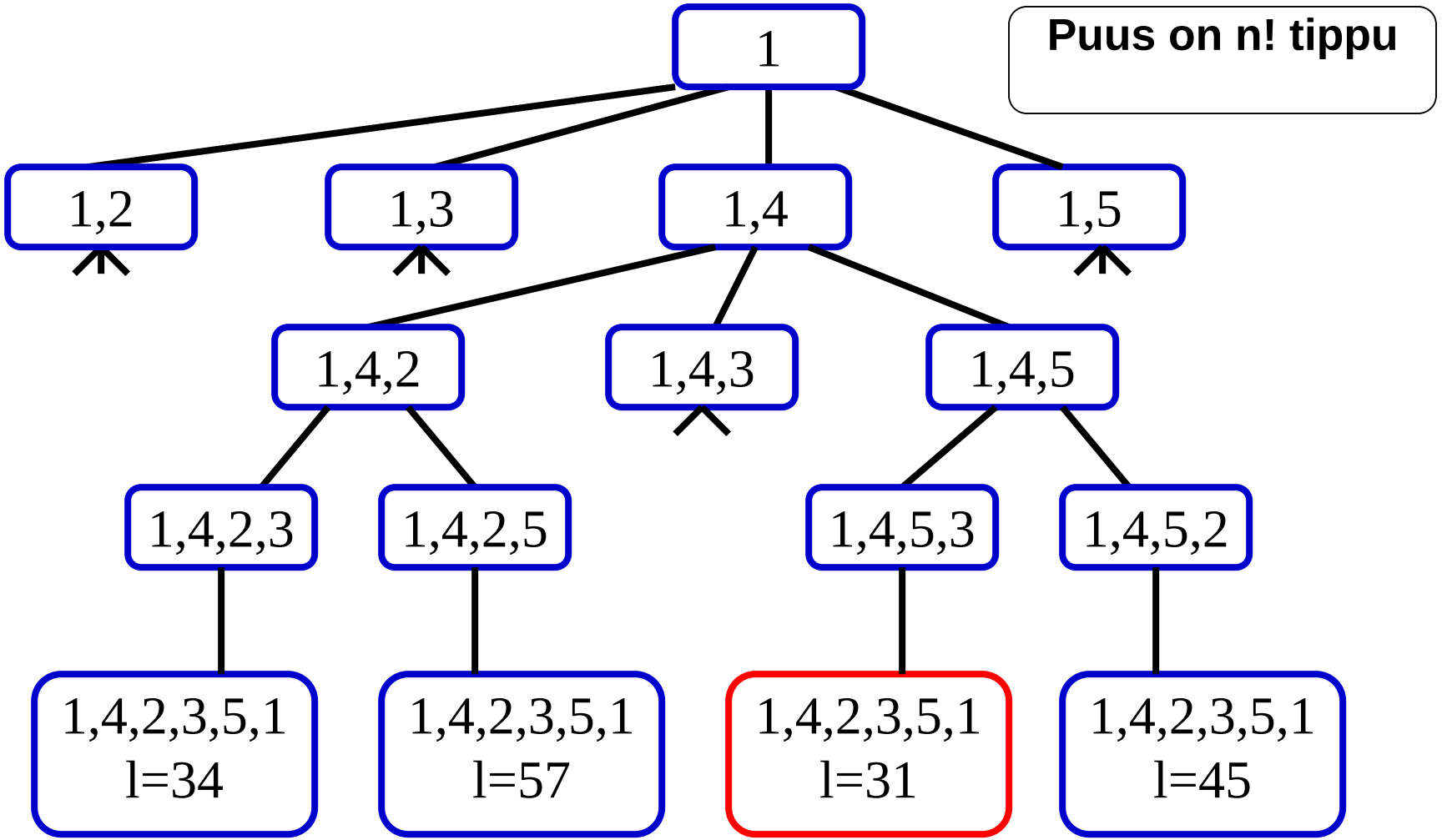
Rändkaupmehe ülesande lahendamine

- Rändkaupmehe ülesanne on klassikaline kombinatoorne optimiseerimisülesanne
- Lahenduste ajalugu

aasta	ülesande suurus
1954	49
1971	64
1980	318
1994	7397
2001	15112
2006	85900



Rändkaupmehe otsingupuu



Rändkaupmehe täielik algoritm

```
int[] läbitud, parim_tee;
int parim_teepikkus;
tsp(int läbitud_arv) {
   if(läbitud_arv == N){
      teepikkus = teepikkus(läbitud);    //koos teega algusse
      if(teepikkus < parim_teepikkus){
           parim_teepikkus = teepikkus;
           parim_tee = läbitud;              // tuleb teha koopia
      }
   else
      for(i:=1; i <= N; i++)
           if(i ∉ läbitud) {
                läbitud[läbitudarv+1] = i;
                tsp(läbitud_arv+1);
           }
}
```

Rändkaupmehe tagasivõtmisega algoritm

```
int läbitud[N];        // läbitud linnade numbrid läbimise järjekorras
int läbimata[N];       // läbimata[i]=0 kui linn i läbimata
int parim_tee[N] = ahne_hinnang();
int parim_teepikkus = teepikkus(parim_tee);
tsp(int läbitud_arv) {
   if(läbitud_arv == N){
      teepikkus = teepikkus(läbitud);        // koos teega algusse
      if(teepikkus < parim_teepikkus){
           parim_teepikkus = teepikkus;
           parim_tee = läbitud;              // tuleb teha koopia
      }
   elif(bound(läbitud, läbimata) < parim_teepikkus)
      for i in läbimata {
           läbitud[läbitud_arv+1] = i;
           läbimata[i] = 1;
           tsp(läbitud_arv+1);
   }
}
```

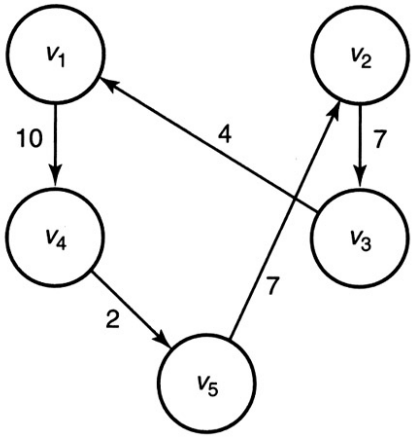
Kärpimine - *bound*()

- hinnang väljendab teepikkust, millest lühemat teed pole antud seisust enam võimalik saavutada
- läbitud teepikkuse ja läbimata linnade läbimise miinimumhinnangu summa
  - igast linnast väljuvate minimaalsete teede järgi
    - arvutatakse igast linnast väljuv minimaalne tee
    - hinnang on summa juba läbitud teest ja minimaalsetest väljuvatest teedest linnade kohta, kust ei ole veel välja sõidetud (sh ka teekonna viimane liińn)
    - minimaalsed teed järgnevas näites [4 7 4 2 4] ja hinnang kui pole läbitud veel ühtegi linna on nende summa 21
  - korrigeeritud minimaalsete teede järgi
    - linnast väljuv minimaalse tee pikkus leitakse ainult läbimata linnade alusel (näide järgmisel slaidil)
  - hinnagu algoritm tuleb implementeerida efektiivselt

Rändkaupmehe otsingupuu kärpimine

- Otsida mingi esialgne tee
  - ahne algoritmiga
  - minimaalse katva puu abil
- Leida teede pikkuse minimaalne hinnang
  - kõigi tippude väikseima kaaluga väljuvate (sisenevate) kaarte kaalude summa
- Kärpimine otsingul
  - kärpimine läbitud tee ja läbimata tippude min hinnangu summa abil
  - otsing tagasivõtmisega või parim enne strateegiaga

					min
0	14	4	10	20	4
14	0	7	8	7	7
4	5	0	7	16	4
11	7	9	0	2	2
18	7	17	4	0	4



Rändkaupmehe ülesande ahne lahendamine

- Ei anna optimaalset lõpptulemust
  - võib kasutada kiire lahenduse saamiseks
  - esialgse hinnangu saamiseks kärpimisele
- Ahne algoritmi idee
  - mine alati lähimasse läbimata linna (näitel tulemuseks 31)
- Variatsioonid
  - min alustades erinevatest linnadest (näitel tulemuseks 30)
  - lõpust ettepoole
  - ühe sammu ettevaatamine
  - jne
- Kasutades minimaalsele katvale puule tuginevat lähendavat algoritmi

Tagasivõtmisega ja H&K optimiseerimisalgoritmi loomine

- Defineeri probleemi olekuruum, mida on vaja läbi otsida
  - mida kujutab endast iga valikupuu tase
  - millised on valikud (harud) igast tipust
  - leia seda puud läbikäiv algoritm
  - sellega on olema täielik (*frute force*) algoritm, mida saab katsetada väikeste ülesannete lahendamiseks
- Leia hindamiskriteerium, millega arvutada parima võimaliku tulemuse tõke vaadeldavast otsingupuu tipust.
  - kasuta saadud hinnangut puu lootusetute harude kärpimiseks
  - H&K algoritmi korral võib kasuta hinnangut ka järgmise vaadeldava tipu valikuks

2. programeerimisülesanne

Leida optimaalne lahendus ATSP (*asymetric travelling salesman*) ülesandele.

Leida lühim tee, mis algab ja lõpeb samas linnas, ja läbib kõiki linnu täpselt üks kord.

Soovitusi

- Alustage ilma kärpimiseta täielikust algoritmist
  - kontrollige, et see töötab
  - lisage kärpimine ja kontrollige, et saate õigeid vastuseid
  - täiendage kärpimismeetodit, kontrollige tulemusi
- Oma programmi testimiseks kasutage kõigepealt väikesi ülesandeid (4-7 linna), mida olete suutelised ka ise paberi peal lahendama
- Veebilehel on viidad testülesannetele koos optimaalsete lahendustega
  - teie leitud lahendus võib olla erinev, aga teepikkus peab olema sama

Kokkuvõtteks

- Tagasivõtmisega ning Hargne&Kärbi algoritme kasutatakse ülesannete puhul, kus on vaja leida optimaalne lahendus
- Sisuliselt täielik algoritm (jõumeetod) heuristilise optimiseerimisega (kärpimisega)
  - täielik - põhineb konfiguratsioone esitava otsingupuu täielikul läbivaatusel
  - heuristika - on võimalik leida meetodeid otsingupuu kärpimiseks
- Otsingupuu ei ole neis algoritmides andmestruktuur vaid algoritmi seisu (konfiguratsiooni) kajastavate väärtuste struktuur, st abstraktsioon, mille abil algoritmi tööd ette kujutada.
  - tagasivõtmisega algoritmis kujutab ka rekursiivseid väljakutseid