# Variantide läbivaatus

On palju ülesandeid, milles küsitakse mingi sisendandmetest sõltuva hulga:

- kõiki elemente,
- suvalist elementi,
- elementide arvu.

Mõnikord on igale elemendile seatud vastavusse väärtus (kaal). Sel juhul võidakse küsida:

- elementide väärtuste summat,
- maksimaalse/minimaalse väärtusega elementi.

Kui ei õnnestu leida efektiivsemat algoritmi, mis kasutaks ära ülesande spetsiifikat, võib lahenduse programmeerida "jõuga" kõikide variantide (ehk hulga elementide) läbivaatuse abil.

## *Variantide genereerimine*

Variante võib genereerida kas iteratiivselt või tagurdusmeetodiga.

Iteratiivse genereerimise korral konstrueeritakse esmalt eraldi üks variant (nn "seeme" või "baas") ja seejärel iga juba leitud variandi muutmise teel talle mingis loogilises järjekorras järgnev. Üldiselt on iteratiivne genereerimine keerulisem programmeerida ja lisaks on seda raskem ülesande spetsiifikast lähtuvalt optimeerida.

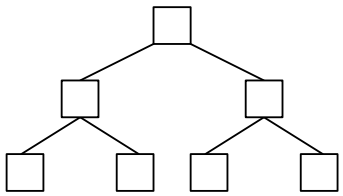
Tagurdusmeetodi (*backtracking*) realiseerimiseks on kõige parem kasutada rekursiooni. Põhiline idee seisneb selles, et variant esitatakse teatud andmestruktuurina, näiteks täisarvuliste muutujate $a_1$, ..., $a_m$ jadana, ning seda andmestruktuuri hakatakse järg-järgult täitma kõikidel võimalikel viisidel. Vaatleme seda muutujate jada näitel:

```
var
     a : array [1..m] of integer;

procedure gen(i : integer);
var j : integer
begin
     if i > m then begin
          if <a on korrektne variandi kirjeldus> then <töötle>;
          exit;
     end;

     for j := <min_a_i> to <max_a_i> do begin
          a[i] := j;
          gen(i+1);
     end;
end;

begin
     gen(0);
end.
```

Siin tähendavad <min_a_i> ja <max_a_i> vastavalt muutuja $a_i$ minimaalset ja maksimaalset lubatud väärtust. Tingimus <a on korrektne variandi kirjeldus> tähendab seda, et kontrollitakse, kas väärtuste jada rahuldab ülesandes seatud piiranguid. Kohas <töötle> tuleks uut varianti

kasutada. Kui ülesandes küsitakse variantide loendit, saab variandi väljastada, kui küsitakse elementide arvu, saab suurendada loenduri väärtust jne.

Üldiselt võib rekursiivset variantide läbivaatamist vaadelda puu sügavuti läbimisena:



Puu juur tähistab rekursiivse alamprogrammi esmakordset väljakutset, iga haru vastab rekursiivse alamprogrammi iseenda väljakutsumisele. Sellise puu lehtedeks on genereeritud variandid. Tihti juhtub nii, et reaalsele variandile ei vasta mitte igasugune selle andmestruktuuri sisu. Selliseid variante pole mõtet vaadelda. Tihti saab sellest aru enne puu leheni jõudmist (kuskil vahetipus), s.t. enne seda kui andmestruktuur on täielikult täidetud. Sellist puu haru pole mõtet edasi vaadelda. Selline otsingupuu pügamine (*search tree pruning*) võimaldab tihti hoida kokku väga palju arvutusi.

## Harude ja tõkete meetod

Oluline puu pügamise tehnika on harude ja tõkete meetod (*branch and bound*), mis on rakendatav juhul, kui otsitakse hulga maksimaalse (minimaalse) väärtusega elementi. Põhiline idee seisneb selles, et iga variantide puu tipu jaoks arvutatakse optimistlik hinnang: väärtus, mis on kindlasti suurem (väiksem) kõikidest reaalsete variantide väärtustest, mis on sellest tipust algava alampuu lehtedes. Nagu alati, läbitakse puud sügavuti. Globaalses mälus hoitakse seni leitud suurima (vähima) väärtusega variandi kirjeldust. Kui mingi tipu optimistlik hinnang osutub väiksemaks (suuremaks) kui seni leitud parima variandi väärtus, siis selle tipu alampuud pole mõtet läbida, kuna kõikide selle puu lehtedes asuvate variantide väärtused on liiga väikesed (liiga suured).

## Nipid

Järgnevas esitame mõned üksikud kasulikud nipid mis on seotud variantide läbivaatusega:

- Kasuta oma fantaasiat. Eeltoodud algoritmid ei ole aksioom. Muuda neid vastavalt vajadustele (aga tõesta oma idee korrektsus).
- Mälu ja aja kokkuhoiu mõttes tasub seisu hoida globaalses mälus (mitte edastada seda alamprogrammi parameetrina). Enne rekursiivset väljakutset peab siis seisu kirjeldust vajalikul moel muutma ja pärast esialgse seisu taastama. Võib öelda, et seisu hoitakse globaalses mälus, aga selle muutusi programmi magasinis.
- Mõnikord osutub vahepealse variandi kontrollimisel, et seda haru pole mõtet edasi vaadelda ja enamgi veel, tagasi tuleb võtta rohkem kui üks samm.
- Kui on raske genereerida õiget hulka, võib katsuda genereerida mõnda teist ja siis selle alusel arvutada vastus õige hulga kohta. Näiteks kui küsitakse erinevate kujundite arvu, kus peegeldusi ja pöördeid loetakse võrdseks, võib
    a. genereerida kõik võimalikud kujundid peegeldusi ja pöördeid arvestamata, lugeda iga kujundi puhul kokku, mitu erinevat kujundit saab, kasutades pöördeid ja peegeldusi ja kasutada seda informatsiooni esialgsele küsimusele vastamiseks;
    b. defineerida reegli, mille järgi igast samaväärsete kujundite grupist loendatakse ainult üks esindaja.

- Kui otsingupuus on palju lehti erineval sügavusel ja meid huvitab juurtipule lähim nõutud omadustega leht, võib puud läbida korduvalt, suurendades maksimaalse lubatava sügavuse piiri seni, kuni leitakse uusi vastuseid. Kuna puu suurus sõltub sügavusest eksponentsiaalselt, siis on tehtavate üleliigsete operatsioonide arv (väiksema sügavusega tippude korduv läbimine) konstantne arv kordi suurem kui viimasel sügavusel tehtavate operatsioonide arv ($1 + 2 + ... + 2^i + ... + 2^n = 2^{n+1}-1$). See-eest ei ole vaja läbida tippe, mis on sügavamal kui otsitav vastus (nende arv on üldiselt palju suurem ja võib olla isegi lõpmatu).
- Tagurdusmeetodi võib realiseerida ka mitterekursiivselt, ilmutatud magasini abil.

## NP

Teoreetilises informaatikas on tähtsal kohal ülesannete klass nimega NP (Non-Deterministically Polynomial). Mitteformaalselt kuuluvad NP-sse kõik ülesanded millel on järgnev kuju:

- Kõik variandid on kirjeldatavad bittide jadana, mille pikkus on tõkestatud polünoomiga sisendi suurusest.
- On antud protseduur, mis kontrollib variandi sobivust polünomiaalse ajaga selle pikkuse suhtes.
- Öelda kas leidub sobiv variant.

Sisuliselt on tegemist suvalise variantide läbivaatusel põhineva ülesandega, kus küsitakse ühte võimalikest sobivatest variantidest. Väga suur osa praktilistest ülesannetest kuuluvad klassi NP.

Öeldakse, et ülesanne A on polünomiaalselt taandatav ülesandele B, kui on olemas algoritm mis teisendab A sisendi B sisendiks, nii et ülesande B vastus on ka ülesanne A vastus. Kusjuures on tähtis, et algoritm töötaks sisendi suuruse suhtes polünomiaalse ajaga. Öeldakse, et ülesanne on NP-täielik (*NP-complete*) kui kõik NP ülesanded on polünomiaalselt temale taandatavad. On leitud sadu selliseid ülesandeid (rändkaupmehe ülesanne, seljakoti pakkimine, klikiülesanne).

Kui õnnestuks leida algoritm, mis lahendaks ühe NP-täieliku ülesande polünomiaalse ajaga, oleks võimalik lahendada kõik NP ülesanded polünomiaalse ajaga. Kahjuks ei ole seni leitud algoritmi, mis oleks põhimõtteliselt kiirem kui variantide läbivaatus. Viimane, nagu teada, töötab eksponentsiaalse ajaga. Asjaga tegeletakse aktiivselt juba peaaegu 30 aastat, kuid seni pole sellist algoritmi leitud. Samas pole ka tõestatud et teda pole olemas.

## Mängupuu, minimaks, $\alpha$-$\beta$-pügamine

Kahe mängija mänge (näiteks trips-traps-trull, male, kabe) on mugav vaadelda puuna: juur vastab algseisule, sellest väljuvad kaared vastavad esimese mängija A käikudele ja viivad seisudesse, mis on saavutatavad ühe käiguga, neist viivad kaared vastavad teise mängija B käikudele jne.

On olemas ülesannete tüüp mille puhul antakse mängu mingi seis, omistatakse mängupuu lehtedele (lõppseisudele) kaalud (mängija võit) ja küsitakse, milline on maksimaalne antud mängija jaoks garanteeritud võit. Kui mängupuu on liiga suur (näiteks male puhul), "lõigatakse" puu mingil sügavusel ära ja omistatakse tekkinud lehtedele maksimaalse garanteeritud võidu hinnangud.

Standardne algoritm, mida kasutatakse sellist tüüpi ülesannete lahendamiseks, on minimaks (*minimax*). Kirjeldame seda algoritmi mängija A jaoks. Minimaksi idee seisneb selles, et kasutatakse kahte rekursiivset alamprogrammi `searchA()` ja `searchB()`. Üks vastab mängija A ja teine tema vastase B sammudele. Mõlemad saavad argumendiks mängu seisu, kus nende mängija peab tegema järgmise käigu ja tagastavad maksimaalse garanteeritud võidu A jaoks.

Alamprogramm `searchA()` teeb kõik võimalikud mängija A sammud, kutsub iga saadud seisu jaoks välja alamprogrammi `searchB()` ja tagastab maksimaalse selle alamprogrammi poolt tagastatud väärtuse. Alamprogramm `searchB()` proovib läbi kõik mängija B sammud, kutsub saadud seisude jaoks välja alamprogrammi `searchA()` ja tagastab minimaalse selle alamprogrammi poolt tagastatud väärtuse. Kutsudes välja alamprogrammi `searchA()` algseisu jaoks, saadakse maksimaalne garanteeritud võit. Lisaks on võimalik teada saada millise avakäigu peab mängija A tegema selleks, et seda võitu saavutada: käigu, mis andis (maksimaalse) väärtuse, mille tagastas alamprogramm `searchA()`.

Minimaksi kiirendamiseks saab kasutada tehnikat nimega α-β-pügamine (*α-β pruning*), mis seisneb selles, et mõlemad alamprogrammid saavad veel kaks argumenti α ja β, mis on minimaalne ja maksimaalne väärtus, mida on mõtet tagastada. Kui tagastatav väärtus on väljaspool seda vahemiku, siis ta "kaob ära" kuskil "üleval" võetavas maksimumis või miinimumis.

```
function searchA(x : State; a , b : integer) : integer;
begin
     for <mängija A iga samm> do begin
          a := max(a, searchB(<järgmine seis>, a, b));
          if a >= b then break;
     end;
     searchA := a;
end;

function searchB(x : State; a, b : integer) : integer;
begin
     for <mängija B iga samm> do begin
          b := min(b, searchA(<järgmine seis>, a, b));
          if b <= a then break;
     end;
     searchB := b;
end;
```

# *Näiteid*

## Hamiltoni tsükkel

Ülesanne: Antud graaf. Leida tsükkel mis läbib kõik tipud täpselt üks kord.

Lahendus:

```
var
     kasut : array [1..N] of boolean;
     ahel : array [1..N] of integer;

procedure search(i : integer);
begin
     if i > N then begin
          if <ahel[N] on ühendatud ahel[1]-ga> then begin
               <lahendus leitud>;
          end;
          exit;
     end;

     for j := 1 to N do begin
          if not kasut[j] and <ahel[i-1] on ühendatud j-ga> then begin
               ahel[i] := j;
               kasut[j] := true;
               search(i+1);
               kasut[j] := false;
          end;
     end;
end;

begin
     ahel[1] := 1;
     kasut[1] := true;
     search(2);
end.
```

## Lühim Hamiltoni tsükkel

Ülesanne: Antud graaf, mille kaartel on pikkused. Leida lühim Hamiltoni tsükkel.

Lahendus: Siin saab kasutada alusena sama algoritmi, lisades harude ja tõkete meetodi. Ehitatava tsükli pikkuse optimistlikuks hinnanguks võiks olla juba kasutusele võetud kaarte pikkused pluss veel läbimata tippudest väljuvate minimaalse pikkusega kaarte summa. Juhul, kui see hinnang on suurem kui seni leitud parim lahend, pole mõtet seda haru enam vaadelda.

## Ristsõna (Lahtine võistlus 2000)

Ülesanne: antud ristsõna mall (millistesse ruutudesse saab tähti kirjutada) ning teatud hulk sõnu. Täita mall nende sõnadega.

Lahendus: Esiteks peab leidma üles positsioonid, kust võib alata uus sõna. Siis võib neid läbida ülevalt alla, vasakult paremale, proovides paigutada sobivad sõnad kas horisontaalselt või vertikaalselt.

## *Ülesandeid*

- IOI'98 Day 2 "Camelot"
- IOI'98 Day 2 "Polygon"
- IOI'95 Day 2 "Letter Game"
- IOI'94 Day 2 "Busses"
- IOI'94 Day 2 "Sectors"

- BOI'99 Day 2 "Mobile" ("Kangisüteem")
- BOI'98 Day 1 "Championship in Basketball"
- BOI'98 Day 2 "Star"
- BOI'97 Day 2 "Figures"

- Valikvõistlus 2004 "Täringu veeretamine"
- Piirkonnavoor 2004 "Rändkaupmees"
- Lahtine võistlus 2003 "Tutvumine"
- Lõppvoor 2003 "Mosaiik" (2 varianti)
- Koolivoor 2002 "Kodeeritud liitmine"
- Koolivoor 2001 "Lipud"
- Lahtine võistlus 2000 "Ristsõna"
- Lõppvoor 2000 "Tehas"
- Valikvõistlus 1996 1. päev "Lipustuv ettur"
- Valikvõistlus 1996 2. päev "Spiraalmõstatus"
- Lõppvoor 1994 "Kasvav sõnade jada"