Loeng 9: Kitsendustega loogiline programmeerimine (*Constraint Logic Programming*)

J.Vain

ITI0211 Loogiline programmeerimine
Sügis 2021

Mis on kitsendustega loogiline programmeerimine (KLP)?

- Kitsendustega programmeerimine (*constraint programming*) üldiselt on **üks** deklaratiivse programmeerimise vorme, kus programm esitatakse kitsenduste kujul, mida otsitav lahend peab rahuldama.

- KLP-d kasutatakse **kombinatoorsete*** ülesannete kirjeldamisel ja lahendamisel:
  - Ressursside planeerimine
  - Ajaplaanide koostamine
  - Logistika ülesanded
  - Mängude lahendamine (sudoku, ristsõnad, kabe)
  - Graafi tippude värvimine
  - …

* - https://www.sciencedirect.com/topics/computer-science/combinatorial-problem

Kombinatoorsed ülesanded

- *Kombinatoorsed ülesanded* on otsustusülesanded, kus lahend on määratud kitsendavate tingimuste hulgaga (otsinguruum).
- Kombinatoorseid ülesandeid lahendatakse algoritmide abil, mis
  - määrab *otsingu alguspunkti otsinguruumis*, s.o. kitsendustes ja sihifunktsioonis esinevate muutujate algväärtustuse (sageli leitakse need juhuvalikuga),
  - annab otsinguruumi järgmise punkti leidmise eeskirja (tavaliselt põhineb see heuristilisel hinnafunktsioonil),
  - itereerib otsingusamme kuni leitakse lahend, mis rahuldab peatumistingimust.
  - Et vältida otsingu stagneerumist, kasutavad kõik otsingualgoritmid alglahendi ja otsingusammu leidmisel randomiseerimist.

KLP SWI-Prologis

KLP-d toetavad SWI-Prologi teegid:

- library(clpfd): *Constraint Logic Programming over Finite Domains*
- library(clpr): *Constraint Logic Programming over Rationals and Reals*[1]

[1] – Teek tuleb laadida Prologi töömällu, enne teegi predikaatide poole pöördumist. Selleks tuleb kasutada vastavalt deklaratsioone

```
:- use_module(library(clpb)). % boolean domain
:- use_module(library(clpfd)).% finite domain
:- use_module(library(clpr)). % rationals and reals
```

KLP lõplikel hulkadel (clpfd)

Teegis clpfd on 2 tüüpi predikaate:

- *Kitsendusi kirjeldava predikaadid*:
  - Esitavad muutujate seoseid lõplikel täisarvude hulkadel;
  - Lahendamisel interpreteeritakse kitsendusi protseduuriga, mis on täisarvuliste avaldiste väärtustamise üldistus
  - Muutujate väärtustamisel toimub väärtuste ülekanne (*propagation*) üle kõigi kitsenduste, kus vastav muutja esineb.
- *Otsinguruumi läbimist suunavad predikaadid* (*enumeration predicates*):
  - Võimaldavad esitada lahendi otsimise strateegiaid muutujate määramispiirkondadel (määramispiirkondade ristkorrutis defineerib otsinguruumi).

Otsinguruumi kitsenduste kirjeldamine: lõplikul määramispiirkonnal määratud tüübid ja avaldised

an integer	- Täisarvuline tüüp
a variable	- Väärtustamata muutuja
`-Expr`	- Unaarne miinus
`abs(Expr)`	- Absoluutväärtus
`Expr + Expr`	- Liitmine
`Expr * Expr`	- Korrutamine
`Expr - Expr`	- Lahutamine
`Expr / Expr`	- Täisarvuline jagamine
`Expr mod Expr`	- Täisarvulise jagamise jääk
`min(Expr,Expr)`	- Kahe avaldise väärtustest väiksema leidmine
`max(Expr,Expr)`	- Kahe avaldise väärtustest suurema leidmine

Otsinguruumi kitsenduste kirjeldamine: Kitsenduspredikaadid täisarvudel

`Expr1 #>= Expr2`	`Expr1` väärtus on suurem või võrdne kui `Expr2` väärtus
`Expr1 #=< Expr2`	`Expr1` väärtus on väiksem või võrdne kui `Expr2` väärtus
`Expr1 #= Expr2`	`Expr1` ja *Expr2* väärtused on võrdsed
`Expr1 #\= Expr2`	`Expr1` ja *Expr2* väärtused ei ole võrdsed
`Expr1 #> Expr2`	`Expr1` väärtus on suurem kui `Expr2` väärtus
`Expr1 #< Expr2`	`Expr1` väärtus on väiksem kui `Expr2` väärtus

Kitsenduspredikaatide

`in/2, #=/2, #\=/2, #</2, #>/2, #=</2` ja `#>=/2`

tõeväärtusi saab clpfd-s samuti interpreteerida kui täisarve 0 ja 1 (***reification***).

Otsinguruumi kitsenduste kirjeldamine: kitsenduste vahelised loogikaseosed

Tähistagu P ja Q kitsenduspredikaate, siis nendega tehtavad loogikatehted interpreteeritakse järgmiselt:

`#\Q`	tõene *iff*	`Q` on väär
`P #\/ Q`	tõene *iff*	`P` või `Q` on tõene
`P #/\ Q`	tõene *iff*	`P` ja `Q` on mõlemad tõesed
`P #<==> Q`	tõene *iff*	`P` ja `Q` on ühesuguse tõeväärtusega
`P #==> Q`	tõene *iff*	`P`-st järeldub `Q`
`P #<== Q`	tõene *iff*	`Q`-st järeldub `P`

iff – „siis ja ainult siis, kui“ ehk „parajasti siis, kui“

Otsinguruumi kitsenduste kirjeldamine: Näiteid kitsenduspredikaatidest

```
?- [library(clpfd)].
```

Päring kitsenduspredikaadiga

```
?- X #> 3.        % Leida täisarvud, mis on suuremad 3st
X in 4..sup.
```

Kitsenduspredikaadi lahend

sup – supreemum ehk määramispiirkonna ülemine raja

inf – infiimum ehk määramispiirkonna alumine raja

```
?- X #\= 20.     % Leida 20st erinevad täisarvud
X in inf..19 \/ 21..sup.

?- 2*X #= 10.    % Leida lin. võrrandi lahend
X = 5.

?- X*X #= 144. % Leida ruutvõrrandi lahendid
X in -12\/12.
```

Otsinguruumi kitsenduste kirjeldamine: Veel näited kitsenduspredikaatidest

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

Võrrandsüsteemi
$\begin{cases} 4x + 2y = 24 \\ x + y = 9 \end{cases}$
mittenegatiivse lahendi leidmine

Kui kitsendused kehtivad mitmele muutujale, viitame kitsenduses nende hulgale

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y #\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.

Kõigile hulga Vs muutujatele kehtivad kitsendused

?- X #= Y #<==> B, X in 0..3, Y in 4..5. % B on tõene parajasti siis, kui X=Y
B = 0,
X in 0..3,
Y in 4..5.

Reified value

Kuidas programmeerida KLP-s?

Üldine stsenaarium:

1. Leida probleemiga seotud muutujad
2. Kirjelda kitsendused (seosepredikaadid muutujate vahel).
3. Siduda kitsendused loogikatehete abil
4. Lahendi otsingustrateegia kirjeldamiseks kasuta olekuruumi läbimise predikaate.

Näide: Krüpto-aritmeetiline mõistatus.

Leia täisarvuline kodeeringu (0..9) tähtedele S, E, N, D, M, O, R, Y nii, et kehtiks aitmeetilne võrrand

SEND + MORE = MONEY,

NB! Iga erineva tähe kood peab olema unikaalne (mitte korduma)

Näide (järg)

- Esitame ülesande SEND + MORE = MONEY CLP(FD)-s:

```
:- use_module(library(clpfd)).
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]):-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
         all_different(Vars),
         S*1000 + E*100 + N*10 + D +
         M*1000 + O*100 + R*10 + E
         #=
         M*10000 + O*1000 + N*100 + E*10 + Y,
      M #\= 0, S #\= 0.    % Vanimad kümnendkohad ei tohi olla 0-d
```

Näide (järg)

- Prolog leiab lahendi kujul, kus osa väärtustusi on määratud üheselt ja osa vahemikus (juhul, kui leiduvad alternatiivsed lahendid):

```
?- puzzle(As+Bs=Cs).
   As = [9, _G4699, _G4702, _G4705],
   Bs = [1, 0, _G4720, _G4699],
   Cs = [1, 0, _G4702, _G4699, _G4744],
   _G4699 in 4..7,
   all_different([9,_G4699,_G4702,_G4705,1,0,_G4720,_G4744]),
   91*_G4699+_G4705+10*_G4720#=90*_G4702+_G4744,
   _G4702 in 5..8,
   _G4705 in 2..8,
   _G4720 in 2..8,
   _G4744 in 2..8.
```

Päringu muutujad As,Bs,Cs unifitseeritakse reegli päises olevate listidega. Vastuses tagastatakse nende listide elementide aadressid

Lahend:
S=9, E=4..7, N=5..8, D=2..8,
M=1, O=0, R=2..8, Y=2..8

Et mitmesest lahendist leida konkreetsed väärtuste kombinatsioonid, tuleb valida ühest väärtuste hulgast konkreetne väärtus ja lahendada ülesanne uuesti asendades kitsenduse konkreetse väärtusega. Itereerida kõikide väärtushulkade korral, mis on saadud eelmisel sammul .

Näide (järg): Kuidas esitada kitsenduste lahendi otsingustrateegiat?

- Kitsenduste lahendaja leiab kõigile muutujatele kas täpsed väärtused või väärtuste vahemikud.
- Selguse huvides on otstarbekas hoida kitsenduste spetsifikatsiooni ja lahendi otsingupredikaate **eraldi**.
- Otsingupredikaadid võimaldavad sama kitsenduste hulga juures rakendada sõltumatult erinevaid strateegiaid.
- Muutujate märgendamisvõte (*labeling*) võimaldab leida ühe kaupa ka konkreetsed lahendid.
- Märgendamine määrab mis muutujatele tuleb leida konkreetne lahend.

Näide (järg)

```
?- puzzle(As+Bs=Cs), label(As).

As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

`label(As)` spetsifitseerib, et listis `As` olevatele muutujatele leitakse konkreetne väärtustus (*term grounding*), mis rahuldab päringus olevaid kitsendusi.

Muutujate määramispiirkonna kitsendused

- Kas muutuja `Var` väärtus on määramispiirkonnas `Domain`?

  `?Var in +Domain`

  - `Domain` võib olla spetsifitseeritud ühel järgmistest kujudest:

    1. `Lower .. Upper`

       Muutuja `Var` väärtus `I` rahuldab tingimust `Lower ≤ I ≤ Upper.`

       Lower on täisarv või aatom **`inf`**, mis tähistab nagatiivset lõpmatust.
       Upper on täisarv või aatom **`sup`**, mis tähistab positiivset lõpmatust.

       `inf` ja **`sup`** tähistavad vastavalt määramispiirkonna alumist ja ülemist tõket.

    2. `Domain1 \/ Domain2`

       Tähistab määramispiirkondade `Domain1` ja `Domain2` ühendit.

Näide: `X in inf..19 \/ 21..sup.`

Mitme muutuja ühise määramispiirkonna kitsenduspredikaat `ins/2`

`+Vars ins +Domain`

- Listis Vars olevate muutujate määramispiirkonnaks on `Domain`.

  Näide: `Vars ins 0..9,`

`indomain(?Var)`

- Muutuja `Var` väärtustatakse tagurdamisel iga kord tema määramispiirkonna erineva väärtusega.
- NB! Muutuja `Var` määramispiirkond peab olema lõplik.

Muutujate märgendamine (*labeling*)

`labeling(+Options, +Vars)`

- Märgendamine tähendab kõigile listi `Vars` muutujatele konkreetse väärtuse omistamist nende määramispiirkonnast.
- Määramispiirkonnad peavad olema eelnevalt defineeritud lõplikud.
- `Options` määrab valikud väärtuste otsingu juhtimiseks määramispiirkondadel (kuidas konkretiseeriv väärtustus leitakse).

Valikud märgendusstrateegia juhtimiseks

`Options ::=`

`leftmost` – märgendab muutujad vastavalt nende järjekorrale listis `Vars` (s.o. suvandi vaikeväärtus).

| **`ff`** – muutujate märgendamise järjekorra määrab nende määramispiirkonna suurus (väiksemalt suuremale järjekorras). See on sageli otstarbekas strateegia, et kiiremini tuvastada ülesande mittelahenduvus.

| **`ffc`** – väiksema määramispiirkonnaga muutujate märgendamine enne; ja prioriteet on muutujatel, mis esinevad rohkemates kitsendustes.

| **`min`** – muutujate märgendamine järjekorras, kus prioriteet on muutujal, mille määramispiirkonna alamraja on vähim.

| **`max`** – muutujate märgendamine järjekorras, kus prioriteet on muutujal, mille määramispiirkonna ülemine raja on suurim.

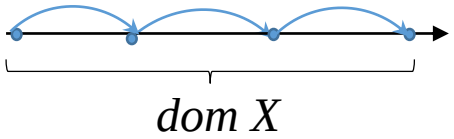
Valikud märgendusstrateegia juhtimiseks (järg)

- Väärtuste läbikäimise strateegiad määramispiirkonna sees:

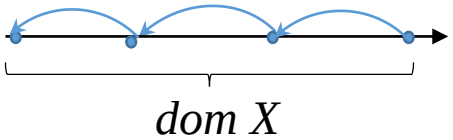
Kui muutujate väärtustamise järjekord on spetsifitseeritud, määratakse väärtuste läbikäimise järjestus nende määramispiirkonna sees.

Väärtuste valimise järjestus on kas

`up` - väärtused valitakse kasvavas järjekorras (vaikimisi).



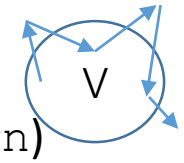
`down` - väärtused valitakse kahanemise järjekorras .



Valikud märgendusstrateegia juhtimiseks

- Väärtuste läbikäimise strateegiad määramispiirkonna sees (järg):

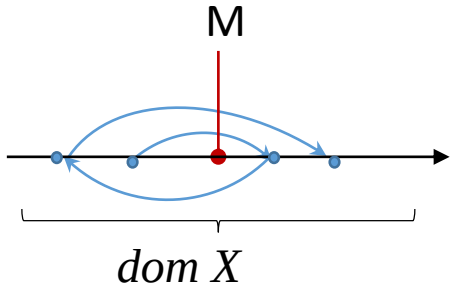
`step` – muutuja `X` väärtustatakse kordamööda määramispiirkonna `V` elemendiga ja elemendiga, mis ei kuulu määramispiirkonda, kusjuures V elemendi valimine on määratud väärtuste järjestussuvandiga (`up/down`)



`enum` - muutuja `X` väärtustatakse järjest määramispiirkonna `V` kõigi elementidega, kusjuures järjekord on määratud järjestussuvandiga.

`bisect` – muutuja `X` väärtus valitakse kordamööda määramispiirkonna keskpunktist `M` alt poolt ja keskpunktist ülalt poolt:

```
X #=< M  and X #> M
```



NB! Igas strateegias saab spetsifitseerida maksimaalselt ainult ühe järjestussuvandi.

Valikud märgendusstrateegia juhtimiseks (järg)

Lahendite otsingu järjestuse spetsifitseerimine:

`min(Expr)` – lahendid genereeritakse aritmeetilise avaldise `Expr` väärtuste kasvamise järjekorras (`Expr` peab sisaldama lahendi muutujaid).

`max(Expr)` – lahendid genereeritakse aritmeetilise avaldise `Expr` väärtuste kahanemise järjekorras (`Expr` peab sisaldama lahendi muutujaid).

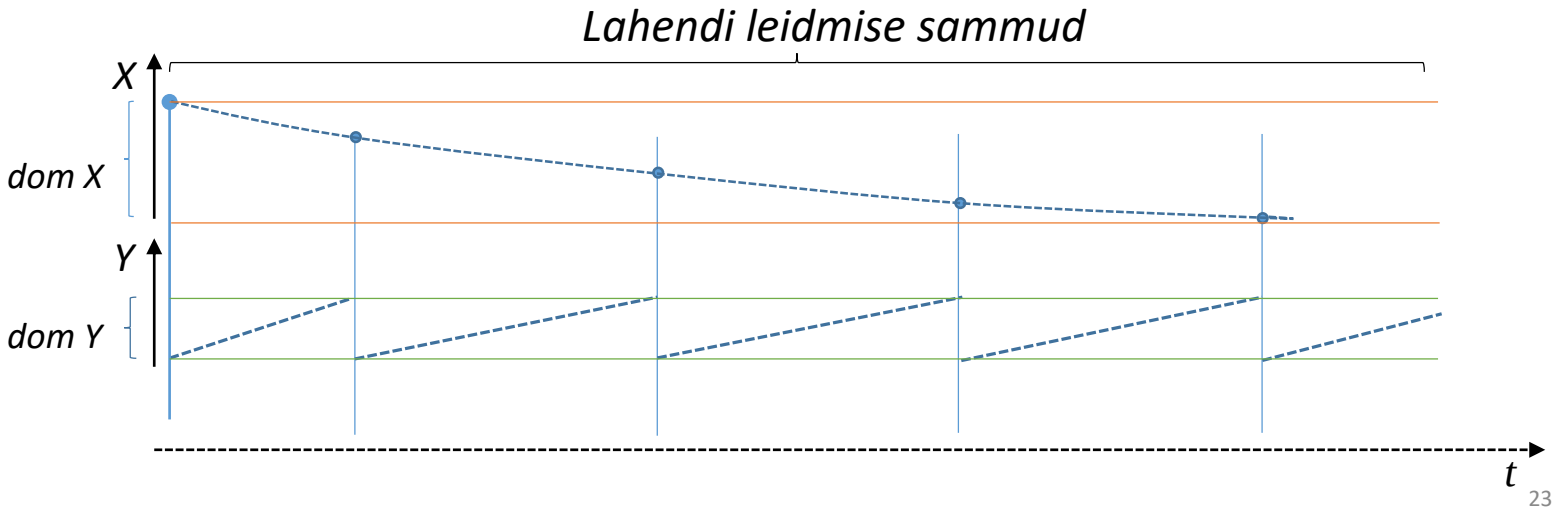
- Märgenduses tuleb näidata kõik `Expr` muutujad, st. need peavad olema konkreetsete väärtustega.
- Kui suvandeid on mitu, siis neid interpreteeritakse vasakult paremale.

Valikud märgendusstrateegia juhtimiseks (järg)

- Näide:

```
?-[X,Y] ins 10..20, labeling([max(X),min(Y)],[X,Y]).
```

  - Muutuja `X` lahendid genereeritakse kahanevas järjekorras,
  - aga `X` iga väärtuse puhul genereeritakse muutuja `Y` väärtused kasvavas järjestuses.



Veel märgendamise võimalusi

`all_different(+Vars)` - muutujad võivad lahendis omada ainult erinevaid väärtusi

`sum(+Vars, +Rel, ?Expr)` - Listi `Vars` elementide summa ja avaldise `Expr` vahel peab kehtima relatsioon `Rel`

Näide:

```
?- [A,B,C] ins 0..sup, sum([A,B,C], #=, 100).
     A in 0..100,
     A+B+C#=100,     % muutjate summa = 100
     B in 0..100,
     C_in_0..100.
```

Veel märgendamise võimalusi

`scalar_product(+Cs, +Vs, +Rel, ?Expr)`

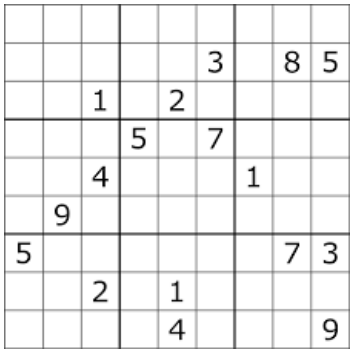
- `Cs` täisarvuliste kordajate list,
- `Vs` muutujate list.

- Tõene, kui `Cs` ja `Vs` skalaarkorrutise ja avaldise `Expr` vahel kehtib seos `Rel`.

- Näide: võrratuse $4a + 5b > a - b$ lahendi leidmine

```
scalar_product([4,5], [A,B], >, A-B).
```

Näide: Sudoku programmeerimine



```
sudoku(Rows) :-
  length(Rows, 9), maplist(length_(9), Rows),        % 9 elementi reas
  append(Rows, Vs), Vs ins 1..9,  % luua 9 rida (list 9st listist), elemendid 1-9
  maplist(all_distinct, Rows),    % iga väärtus reas on unikaalne
  transpose(Rows, Columns),       % transponeerida read veergudeks
  maplist(all_distinct,Columns),  % iga väärtus veerus on unikaalne
  Rows = [A,B,C,D,E,F,G,H,I],     % tähistame read muutujatega
  blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).
                                  % nimega elementidest moodustub 9 plokki
```

<u>Kommentaar:</u>

`maplist(:Goal,?List)` tagastab `true`, kui kõik Listi elemendid rahuldavad Goal'i.

`maplist(:Goal, ?List1, ?List2)` tagastab `true`, kui kõik sama indeksiga elementide paarid listides List1 ja List2 rahuldavad Goal'i

Sudoku programmeerimine (järg)

- `transpose(+Matrix, ?Transpose).` % Transponeerib listide listi

- Näide:

```
?- transpose([[1,2,3],[4,5,6],[7,8,9]], Ts).
   Ts = [[1,4,7], [2,5,8], [3,6,9]]
```

Sudoku programmeerimine (järg)

```
blocks([], [], []).
blocks([A,B,C|Bs1],[D,E,F|Bs2],[G,H,I|Bs3]):-
    all_distinct([A,B,C,D,E,F,G,H,I]),
    blocks(Bs1, Bs2, Bs3).
```

% Defineerime rekursiivselt kõik 3 x 3 väljad nn plokid, kus

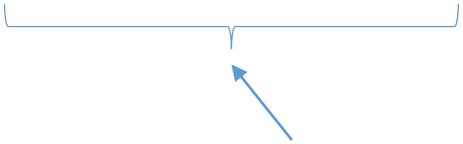
% iga ploki sees kehtib ka unikaalsuse nõue

Sudoku programmeerimine (järg)

```
problem(1,                  % ülesande defineerime faktiga, kus
[[_,_,_,_,_,_,_,_,_],       % väljad on väärtustatud osaliselt
[_,_,_,_,_,3,_,8,5],
[_,_,1,_,2,_,_,_,_],
[_,_,_,5,_,7,_,_,_],
[_,_,4,_,_,_,1,_,_],
[_,9,_,_,_,_,_,_,_],
[5,_,_,_,_,_,_,7,3],
[_,_,2,_,1,_,_,_,_],
[_,_,_,_,4,_,_,_,9]]).
```

Sudoku programmeerimine (järg): päring

```
?- problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
```



Väljastada lahend ridade kaupa

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
[2, 4, 6, 1, 7, 3, 9, 8, 5]
[3, 5, 1, 9, 2, 8, 7, 4, 6]
[1, 2, 8, 5, 3, 7, 6, 9, 4]
[6, 3, 4, 8, 9, 2, 1, 5, 7]
[7, 9, 5, 4, 6, 1, 8, 3, 2]
[5, 1, 9, 2, 8, 6, 4, 7, 3]
[4, 7, 2, 3, 1, 9, 5, 6, 8]
[8, 6, 3, 7, 4, 5, 2, 1, 9]
```

```
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

Näiteid KP-s programmeeritud arvutimängudest

Mängude täielik lahendamine on keeruline!

- Inglise kabe (Checkers) – mäng 8×8 ruudul
- Loodi kabeprogramm Chinook,
- Mängude arv 10^{14}, mis mängiti läbi 18 aasta jooksul.
- Selleks kasutati paralleelselt 50 - 200 lauaarvutit.
- Kõigi käiguvariantide genereerimiseni jõuti 29. apr. 2007
- Meeskonda juhtis Jonathan Schaeffer,
- Tõestati, et parim võimalik tulemus Chinook'i vastu mängides on viik
- Tänaseni on inglise kabe suurima otsinguruumiga lahendatud mäng - 5×10^{20} käiguvarianti