# Otsingustrateegiate programmeerimine

Jüri Vain

ITI0021

# Otsing olekusiirde graafidel

- Näide (olekud ja olekusiirded):
  - Nuppude asetus kabelaual on kabe mängu seisu kirjeldav *olek*
  - Nuppude liigutamine on *siire* ühest olekust teise
- Planeerimisprobleemi lahendamine on *tee* (siirete jada) leidmine, mis viib antud *algolekust* soovitud *lõppolekusse*.

```
% olekud:
state(vabaduse_väljak).
…
state(akadeemia_tee).

% siirded:
move(sõpruse,troll(sõpruse,keemia)).
…
move(estonia,buss(estonia,kaubamaja)).

final_state(akadeemia_tee).
```

# Lahendustee sügavuti otsing (dfs)

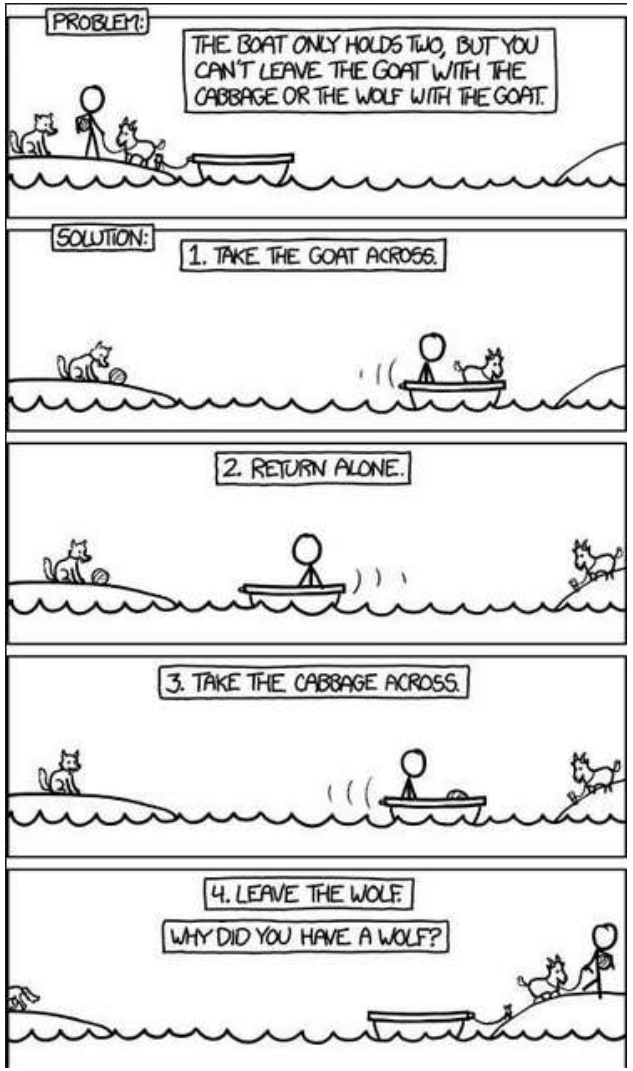
```
solve_dfs(State, History,[]):-
      final_state(State).                  % kas saavutatud olek on lõppolek?
solve_dfs(State, History,[Move|Moves]):-
      move(State,Move),                    % kas olekust leidub siirdeid?
      update(State, Move,State1),          % leia siirde sihtolek
      legal(State1),                       % sihtolek rahuldab kitsendusi?
      not member(State1,History),          % sihtolek ei ole enne läbitud?
      solve_dfs(State1,[State1|History], Moves). % korda otsingut!

?- solve_dfs(estonia,[estonia],-Moves).  % Päring
Moves = [estonia,vabaduse_väljak,… ]
```

# Kuidas defineerida state/3, move/2, legal/1 predikaate?

- Näide: mees, hunt, kits ja kapsas

Olek:

```
state(Boat, LeftBank, RigthBank).
% Boat – paadi asukoht: vasak, parem
% LeftBank – asjad, mis on vasakul kaldal
% RightBank – asjad, mis onparemal kaldal
% Algolek – state(vasak,[kits,hunt,kapsas],[]).
% Lõppolek – state(parem,[], [kits,hunt,kapsas]).
```



# Kuidas defineerida predikaate state/1, move/2,update/2, legal/1?

```
move(state(left,L,R),Cargo):- member(Cargo,L). % vedada saab asju, mis on antud
move(state(right,L,R),Cargo):- member(Cargo,R).  % kohas
move(state(B,L,R),üksi).          % mees sõidab üksi

update(state(B,L,R),Cargo,state(B1,L1,R1):-
      update_boat(B,B1), update_banks(Cargo,B,L,R,L1,R1).

update_boat(vasak,parem).
update_boat(parem,vasak).

update_banks(üksi,B,L,R,L,R).
update_banks(Cargo,vasak,L,R,L1,R1):- select(Cargo,L,L1), insert(Cargo,R,R1).
update_banks(Cargo,parem,L,R,L1,R1):- select(Cargo,R,R1), insert(Cargo,L,L1).
```

# Kuidas defineerida predikaate state/1, move/2,update/2, legal/1?

```
legal(state(vasak,L,R)):- not illegal(R).
legal(state(parem,L,R)):- not illegal(L).

illegal(Bank):- member(hunt,Bank), member(kits,Bank).
illegal(Bank):- member(kits,Bank), member(kapsas,Bank).

insert(El,List,List1):- sort([El|List],List1).

select(El,[El|L],L).
select(El,[El1|L],[El1|L1]):- select(El,L,L1).
```

# Kuidas lahendada suuri ülesandeid?

- Suuremad ülesanded (kabe, male, bridž) eeldavad väga suure olekuruumi läbivaatamist, mis ei ole piiratud lahendusaja korral teostatav.
- Üks võimalik lahendus on piirata käikude hulka kasutades nn *kasufunktsiooni* (Prologis predikaat `value(State, Value)).`
- Levinud strateegiad, mis kasutavad kasufunktsiooni:
  - Mäkkeronimine (*hill climbing*)
  - „parim-esmalt“ otsing (*best-first search*)

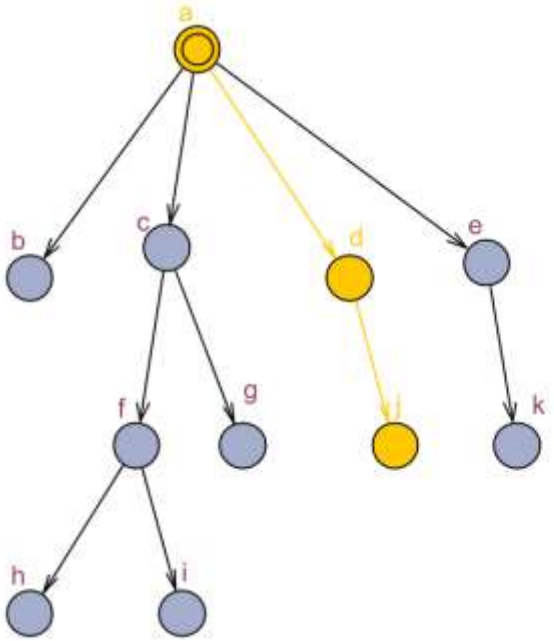
# Otsingustrateegia „*hill climbing*“

- Sügavuti otsingu (*dfs*) üldistus, kus hargnemisel ei valita vasakult järgmine läbimata haru vaid haru, millel on suurim kasufunktsiooni väärtus.
- Kasutame üldist *dfs*-algoritmi, kus `move`
  - genereerib kõik antud olekust ühe siirdega saavutatavad olekud,
  - järjestab need olekud kasufunktsiooni kahanevas järjekorras

# Otsingustrateegia „*hill climbing*“

```
solve_hill_climb(State, History,[]):- final_state(State).
solve_hill_climb(State, History,[Move|Moves]):-
        hill_climb(State,Move),
        update(State,Move,State1),
        legal(State1),
        not member(State1, History),
        solve_hill_climb(State, [State1|History],Moves).

hill_climb(State, Move):-
        findall(M,move(State,M),Moves),
        evaluate_and_order(Moves, State,[],MVs),
        member((Move,Value),MVs).
```

# Näide lahendusest

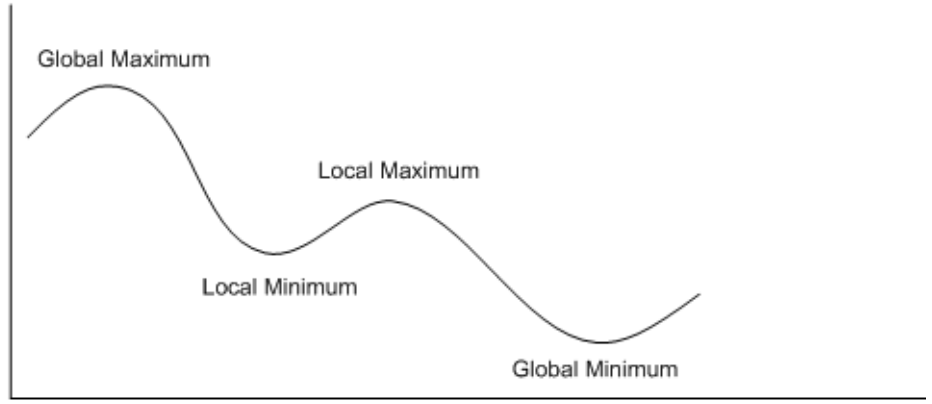



```
move(a,b).
move(a,c).
move(a,d).
move(a,e).
move(c,f).
move(c,g).
move(f,h).
move(f,i).
move(d,j).
move(e,k).
```

```
value(b,1).
value(c,5).
value(d,7).
value(e,2).
value(f,4).
value(g,6).
value(j,9).
value(k,1).
value(h,3).
value(i,2).
```

# Otsingustrateegia „*hill climbing*“

- Sobib juhul, kui
  - on üks sidus olekudiagramm
  - leidub üks parim tee, st. kasufunktsioon annab ühese eelistuse
- otsing on lokaalne st põhineb ainult vahetult saavutatavate olekute võrdlemisel
- Ei sobi globaakse optimumi leidmiseks, kui on palju lokaalseid optimume



# Otsingustrateegia „parim-esmalt“

- Sarnane *hill climbing’*le
- Sügavuti otsimise asemel kasutab *laiuti otsimist* ja *hinnafunktsiooni*
- Hulk *front* sisaldab olekuid, milleni on otsingu käigus jõutud
- Järgmise siirde valimisel vaadatakse kõiki hulgas front sisalduvaid olekuid ja nendest lähtuvaid siirdeid.
- `state(State,Path,Value)`
  - `Path` olekusse jõudmise tee
  - `Value` hinnafunktsiooni väärtus kohal `State`

```
solve_best([state(State,Path,Value)|Frontier], History, Moves):-
      final_state(State), reverse(Path,[],Moves).
solve_best([state(State,Path,Value)|Frontier], History,FinalPath):-
      findall(M,move(State,M),Moves),
      update_frontier(Moves,State,Path,History,Frontier,Frontier1),
      solve_best(Frontier1,[State|History], FinalPath).

update_frontier([M|Ms],State,Path,History,F,F1):-
      update(State,M,State1),
      legal(State1),
      value(State1,Value),
      not member(State1,History),
      insert((State1,[M|Path],Value),F,F0),
update_frontier([],S,P,H,F,F).
```

# Otsing mängude puul

- Vaatame 2 mängija mängusid (kabe, male, tik-tak, jne)
- mängija käik on siire mängu olekute (-seisude) vahel
- käikusid tehakse kas korda-mööda või oleneb käiguõigus eelmise käigu tulemusest
- kummalgi mängijal on oma hinnafunktsioon, mida ta püüab käigu tulemusena maksimeerida (0-summa mängu korral f1=-f2)
- esimese käigu õigus oleneb mängust (valged alustavad, esimene käik loositakse jne)

# Näide

```
play(Game):-
        initialize(Game,Position, Player),      % määrab mängu algseisu
        display_game(Position,Player),          % mänguseisu kuvamine
        play(Position,Player,Game).             % käikude jada käivitamine

play(Position,Player,Result):-                  % Mängu lõpetamistingimus
        game_over(Position,Player,Result),!, announce(Result).
play(Position,Player,Result):-
        choose_move(Position,Player,Move),      % Mängija valib käigu
        move(Move,Position,Position1),          % Käigu sooritamine
        display_game(Position1,Player),
        next_player(Player,Player1),!,          % järgmise käiguõiguse otsus
        play(Position1,Player1,Result).
```

# Mängupuu

- Mängupuu läbimine on sarnane kasufunktsiooniga olekutepuu läbimisele.
- Mängupuud on reeglina liiga suured täieliku otsingu tegemiseks
- Mitte-täieliku otsingu strateegiad:
  - minmax (mängija maksimeerib, vastane minimeerib mängija kasufunktsiooni)
  - mängupuu alfa-beeta kärpimine
- Strateegiat rakendatakse predikaadis `choose_move/3`

# Strateegia rakendamine

```
choose_move(Position,Player,Move):-
      findall(M,move(Position,M), Moves),
      evaluate_and_choose(Moves,Position,(nil,-1000),Move).
```

- `move(Position,Move)` peab olema antud seisus teostatav käik
- `evaluate_and_choose(Moves,Position,Record,BestMove)` tagastab parima käigu
- `Record` – antud seisust parima seni leitud käigu skoor

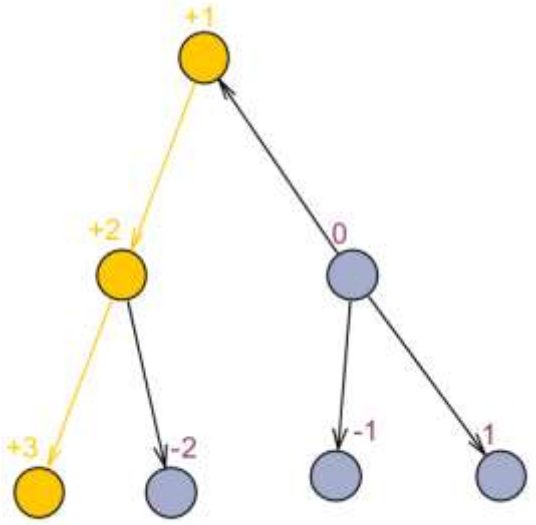
# minmax algoritm

- seisu kasufunktsiooni arvutamiseks arvutame mängu võimalikku seisu mitu sammu ettepoole.
- algoritm eeldab, et vastasmängija valib oma käikudest niisuguse, mis maksimeerib tema kasufunktsiooni ja minimeerib antud mängija oma.
- olenevalt planeerimise sügavusest (mitu käiku ette)

  leitakse  viimase vaadeldava sammu skoori alusel

  kavandatava käigu kasufunktsioon.
- flag – kas antud käigul maksimeerida või minimeerida

  kasufunktsiooni



```
?- evaluate_and_choose(Moves,Position,Depth,Flag,Record,BestMove).

evaluate_and_choose([Move|Moves],Position,D,MaxMin,Record,Best):-
        move(Move,Position,Position1),
        minmax(D,Position1,MaxMin,MoveX,Value),
        update(Move,Value,Record,Record1),
        evaluate_and_choose(Moves,Position,D,MaxMin,Record,Record1,Best).
evaluate_and_choose([],Position,D,MaxMin,Record,Record).

minmax(0,Position,MaxMin,Move,Value):-
        value(Position,V),
        Value is V*MaxMin.
minmax(D,Position,MaxMin,Move,Value):-
        D>0,
        findall(M,move(Position,M),Moves),
        D1 is D-1,
        MinMax is -MaxMin,
        evaluate_and_choose(Moves,Position,D1,MinMax,(nil,-1000),(Move,Value)).
```

# alfa-beeta kärpimine

- Idee: kui selgub, et antud mänguseisust vaadeldava haru jätkamisega ei ole võimalik jõuda seni teadaoleva parima tulemuseni, siis otsing seda haru pidi lõpetatakse.
- http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
- parameetrid:
  - alfa – suurim alamtõke lahendile (alumine läviväärtus maksimeerivale tipule)
  - beeta – vähim ülatõke lahendile (ülemine läviväärtus minimeerivale tipule)
- Otsing minimeerival sammul peatatakse, kui väärtus < alfa
- Otsing maksimeerival sammul peatatakse, kui leitud väärtus > beeta
- Otsingut jätkatakse harupidi, kui $\alpha \leq$ hind $\leq \beta$

# Algoritm

```
evaluate_and_choose([Move|Moves],Position,D,Alpha,Beta,Move1,BestMove):-
       move(Move,Position,Position1),
       alpha_beta(D,Position1,Alpha,Beta, MoveX,Value),
       Value1 is -Value,
       cutoff(Move,Value1,D,Alpha,Beta,Moves,Position,Move1,BestMove).
evaluate_and_choose([],Position,D,Alpha,Beta,Move,(Move,Alpha)).

alpha_beta(0,Position,Alpha,Beta,Move,Value):-
       value(Position,Value).
alpha_beta(D,Position1,Alpha,Beta,Move,Value):-
       findall(M,move(Position,M),Moves),
       Alpha1 is -Beta,
       Beta1 is -Alpha,
       D1 is D-1,
       evaluate_and_choose(Moves,Position,D1,Alpha1,Beta1,nil,(Move,Value)).
```

# Kärpimine

```
cutoff(Move,Value1,D,Alpha,Beta,Moves,Position,Move1,(Move,Value)):-
      Value >= Beta.
cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove):-
      Alfa < Value, Value < Beta,
      evaluate_and_choose(Moves,Position,D,Value,Beta,Move,BestMove).
cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove):-
      Value < Alpha,
      evaluate_and_choose(Moves,Position,D,Alpha,Beta,Move1,BestMove).
```

# Kodutöö

**Ülesande kirjeldus**

- Koostada Prologis kabeprogramm, mis sooritab korraga ühe käigu/võtmise(d).
- Programm peab võistlema vastase programmiga.
- Predikaat “arbiiter” annab programmidele korda-mööda käiguõiguse.
- Mäng lõpeb, kui ühel mängijatest ei ole enam võimalik teha käike. Võitja on programm, mis sooritas viimase käigu.
- Arbiiter kontrollib käikude õigsust ja diskvalifitseerib sohki teinud programmi.

# **Kõik** programmid peavad järgima järgmisi kokkuleppeid:

- 1. Kabelaua seis esitada faktidega ruut/3:

```
ruut(X,Y, Status).    % kus       X,Y ∈ [1,8]

Status = 0        % tühi
Status = 1        % valge
Status = 2        % must
Status = 10       % valge tamm
Status = 20       % must tamm
```

NB! Valged alustavad väiksemate X-koordinaadi väärtusega ruutudest, mustad – suuremate X-koordinaadi väärtusega ruutudest st. valge nupu jaoks leidub algseisus fakt

```
ruut(1,1,1).
```

# Mängija programmi vormistamise reeglid

- Käiku sooritav programm peab olema vormistatud **mooduli** kujul

```
:- module(mooduli_nimi, mooduli_peapredikaat/1).
```

- Mooduli peapredikaat peab olema kujul

```
mooduli_peapredikaat(Color).
```

`Color` –nuppude värv, millega antud programm mängib.

- Moodulis ei tohi esineda staatilisi fakte `ruut/3` ja definitsiooni `:- dynamic ruut/3.`
- Mängija programmist ei tohi väljuda fail-ga st. tagurdamine ei tohi minna teise mängija programmi.
- Soovitav kasutada mängija peapredikaadi järgmist struktuuri:

```
mooduli_peapredikaat(Color):-
                ...., !.
mooduli_peapredikaat(_).
```

# Arbiiteri kohandamine programmidele

- Faktis `players_turn(1,2,Nimi1).` tuleb 3nda parameetri väärtuseks kirjutada programmi nimi, mis mängib MUSTADE nuppudega.

- Fakti `players_turn(2,1,Nimi2).` tuleb 3nda parameetri väärtuseks kirjutada programmi nimi, mis mängib VALGETE nuppudega.

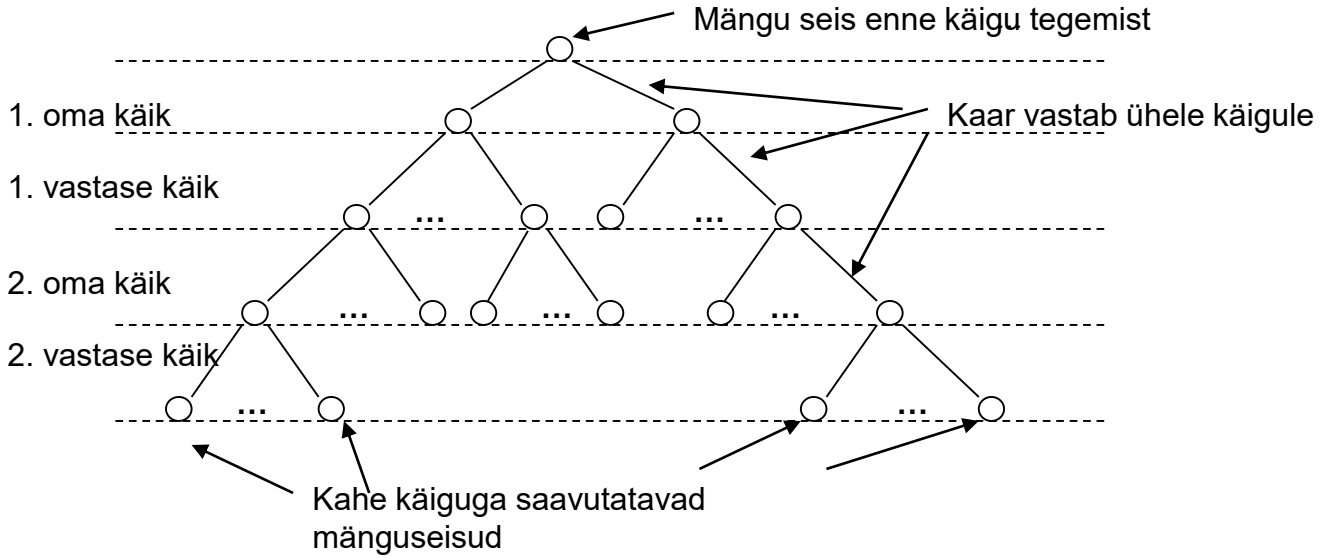
# Mängu käivitamine

- Laadida mällu programm `arbiter.pl`
- Laadida mällu mängijate programmid
- Teha päring mängu käivitava predikaadiga

```
?- turniir.
```

# Abistavaid näpunäiteid käikude planeerimisel

- Käikude planeerimiseks on otstarbekas genereerida saavutatavate mänguseisude puu.
- Igale mänguseisule vastab puu üks tipp ja igale käigule kaar tippude vahel.
- Puu sügavus on määratud sellega kui mitu sammu antud käiku ette planeeritakse.




Joonis 1. Käikude planeerimispuu

# Käigu sooritamiseks vajalikud planeerimistegevused

- Faktide loomine, näiteks musti ruute iseloomustav abifakt `ruut/7` omab järgmist vormingut:

```
ruut(X,Y,Color,Plan_step,Prev_state, Present_state,Cost).
```

kus

- `X,Y` – ruudu koordinaadid [1,..,8]
- `Color` -- ruudul oleva nupu värv [1,2]
- `Plan_step` -- planeerimispuu tase vahemikus [0,..,n], millele fakt vastab
- `Prev_state` -- eelmise seisu ID, millest tekib jooksev seis
- `Present_state` -- käigu tulemusena tekkiva seisu ID
- `Cost` -- käigu tulemusena tekkiva seisu hind. Näiteks vastase nupu võtmisel: Cost:=Cost+1,
  oma nupu kaotamisel: Cost:=Cost-1

# Planeerimispuu genereerimine ja läbimine

- Planeerimispuu koosneb `ruut/7` faktidest, mille parameeter `Prev_state` võimaldab puud läbida terminal-tipust juur-tipu suunas.

- Parima käigu valimine:
  - Kasutades fakti `ruut/7` parameetri `Cost` väärtusi, leida planeerimispuu terminaalsetele tippudele vastavate (, kus parameeter Plan_step = max planeerimissügavus, näiteks `Plan_step = 2`) faktide `ruut/7` hulgast niisugune, mille parameeter `Cost` omab suurimat väärtust.
  - Kasutades fakti `ruut/7` parameeterit `Prev_state` liikuda planeerimispuu juurtipuni ja kuulutada sellele teele jääva esimese käigu tulemus mängu uueks seisuks.
- Kopeerida valitud käiguga tekkiv uus seis vastasele nähtavaks faktide hulgaks `ruut/3` ja anda juhtimine tagasi arbiiterprogrammile.