Loeng 10: Otsingustrateegiate programmeerimine

Jüri Vain

ITI0211

Otsing olekusiirde graafidel

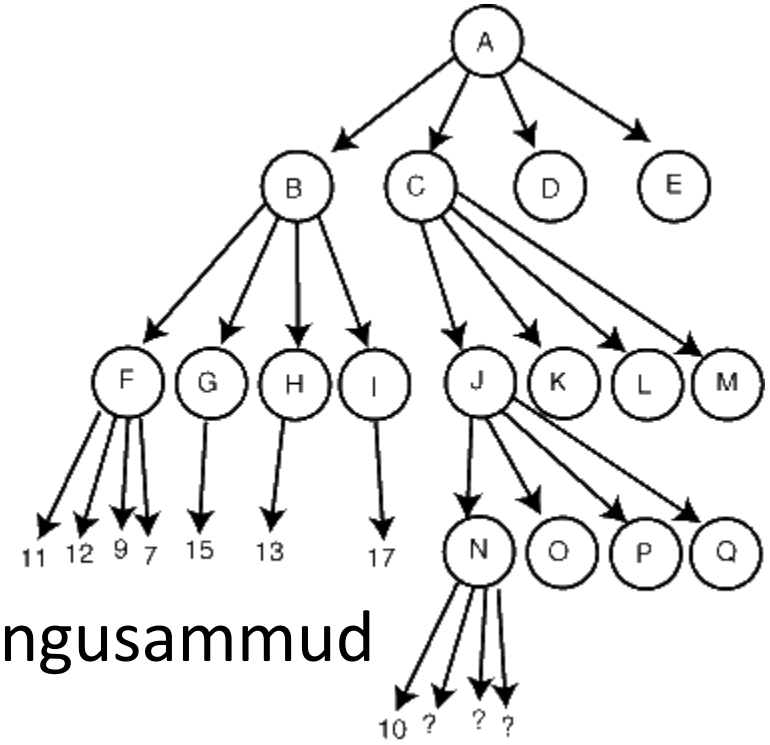
- Planeerimisülesande kirjeldus annab olekute hulga, olekute vahelised siirded, alg- ja lõppoleku.
- Planeerimisprobleemi lahendiks on *tee* (siirete jada), mis viib antud *algolekust* soovitud *lõppolekusse* nii, et teele seatud *lisakitsendused* oleks täidetud.
- **Näide 1**: kabe. Nuppude asetus kabelaual on mängu seisu kirjeldav *olek*, nuppude asendi muutus on *siire* ühest olekust teise.

Näide 2: logistika ülesanne

```
% predikaat "state" näitab sõiduki asukohta
state(vabaduse_väljak).
…
state(akadeemia_tee).
% predikaat"move" esitab transpordiühendusi
move(sõpruse,troll(sõpruse,keemia)).
…
move(estonia,buss(estonia,kaubamaja)).
final_state(akadeemia_tee).
```

Otsingupuu

- Juurtipp on algolek, millest otsing lähtub
- Puu tippudeks on otsingu vaheolekud
- Terminaltippudeks on lõpp- ehk sihtolekud
- Tippudevahelisteks kaarteks on atomaarsed otsingusammud (olekutevahelised siirded).



Lahenduse sügavuti otsing (dfs)

```
solve_dfs(State, History,[]):-
      final_state(State).              % kas jooksev olek on lõppolek?
solve_dfs(State, History,[Move|Moves]):-
      move(State,Move),     % kas olekust leidub siire edasiminekuks?
      update(State, Move,State1),    % leia siirde sihtolek
      legal(State1),              % kas sihtolek rahuldab kitsendusi?
      not member(State1,History), % kas sihtolek läbitakse 1st korda?
      solve_dfs(State1,[State1|History], Moves). % otsing uuest olekust

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

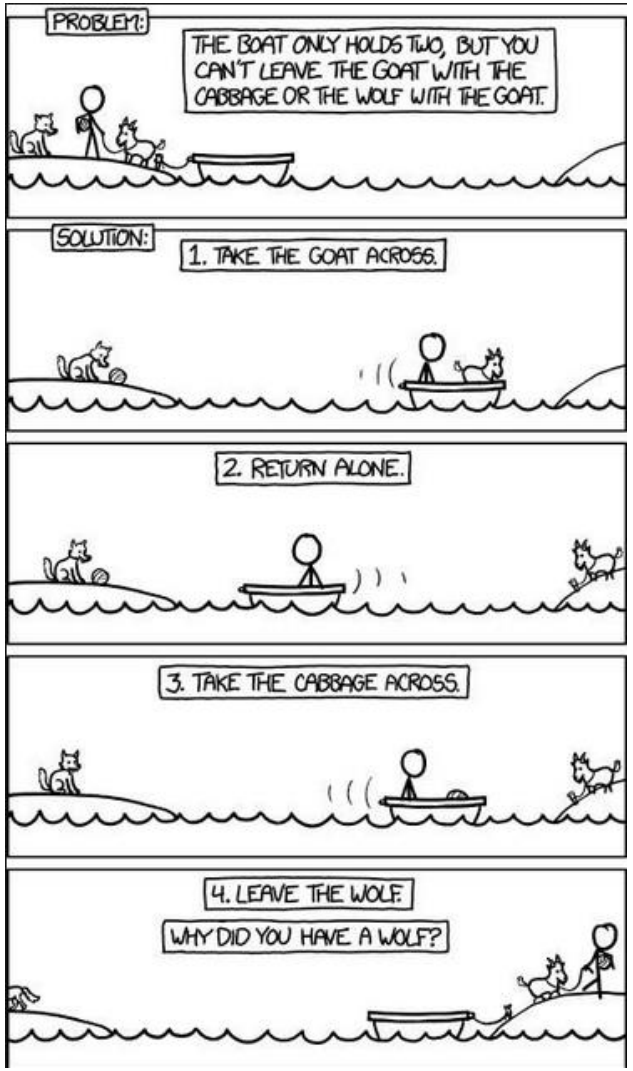
Kuidas defineerida predikaate state/3, move/2, legal/1?

- Näide:

Objektid: mees, hunt, kits ja kapsas

Süsteemi olek: **`state`**`(Boat, LeftBank, RigthBank).`

```
% Boat – paadi asukoht: vasak, parem
% LeftBank – asjade list, mis on vasakul kaldal
% RightBank – asjade list, mis on paremal 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,LB,_),Cargo):- member(Cargo,LB).      % vedada saab asju, mis on
move(state(right,_,RB),Cargo):- member(Cargo,RB).     % samal kaldal paadiga.
move(state(_,_,_),üksi).                              % mees sõidab üksi

update(state(B,LB,RB),Cargo,state(B1,LB1,RB1):-       % uuenda paadi ja kallaste
      update_boat(B,B1), update_banks(Cargo,B,LB,RB,LB1,RB1). % olekut

update_boat(vasak,parem).                  % paadi oleku uuendamine
update_boat(parem,vasak).
```

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

```
%-------------Kallaste olekute uuendamine---------------
update_banks(üksi,_,L,R,L,R).              % kallaste olek ei muutu
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).

%------------ Removing from list -----------------
select(El,[El|L],L).                          %
select(El,[El1|L],[El1|L1]):- select(El,L,L1).

%------------ Inserting to list -------------------
insert(El,List,List1):- sort([El|List],List1). % Inserting to list
```

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

```
% kui paat mehega ühel kaldal, siis teisel kaldal ei tohi olla keelatud paare
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).
```

Kuidas lahendada suuri ülesandeid?

- Suuremad planeerimisülesanded (kabe, male, bridž,…) eeldavad väga suure olekuruumi läbivaatamist, mis ei ole piiratud lahendusaja korral sageli teostatav.
- Üks võimalik lahendus on piirata käikude hulka kasutades nn *kasufunktsiooni*:

$$Gain\_function: State \rightarrow Value$$

(Prologis predikaat `value(State, Value)).`

- Levinud on otsingustrateegiad, mis kasutavad kasufunktsiooni:
  - *hill climbing* - mäkketõus
  - *best-first search* – esmalt-parim

Otsingustrateegia „*hill climbing*“

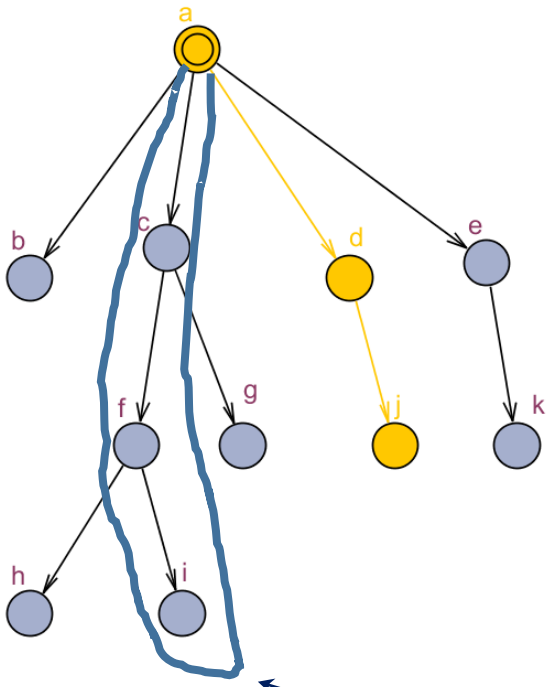
- Sügavuti otsingu (*dfs*) üldistus, kus hargnemisel ei valita vasakult järgmine läbimata haru vaid haru, millel on suurim kasufunktsiooni väärtus.
- Kasutame üldist *dfs*-reeglit, kus predikaadi `move(State,Move)` asendame predikaadiga `hill_climb(State,Move)`.
- `hill_climb(State,Move)`:
  - genereerib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

Otsingustrateegia „*hill climbing*“

```
solve_hill_climb(State, History,[]):- final_state(State).
solve_hill_climb(State, History,[Move|Moves]):-
      hill_climb(State,Move),          % Asendame siin predikaadi 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), % Leia olekust võimalikud sammud
      evaluate_and_order(Moves, State,[],MVs), % Järjesta parim esimene
      member((Move,Value),MVs).   % järjestatud hulgast esimene on parim
```

Näide strateegia *hill climbing* rakendamisest




```
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(h,1).
value(i,9).
value(j,1).
value(k,2).
```

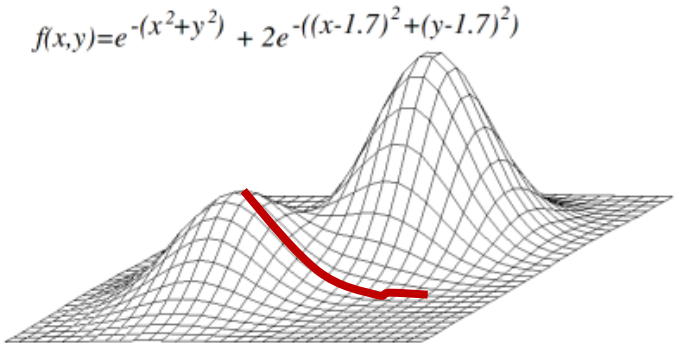
8

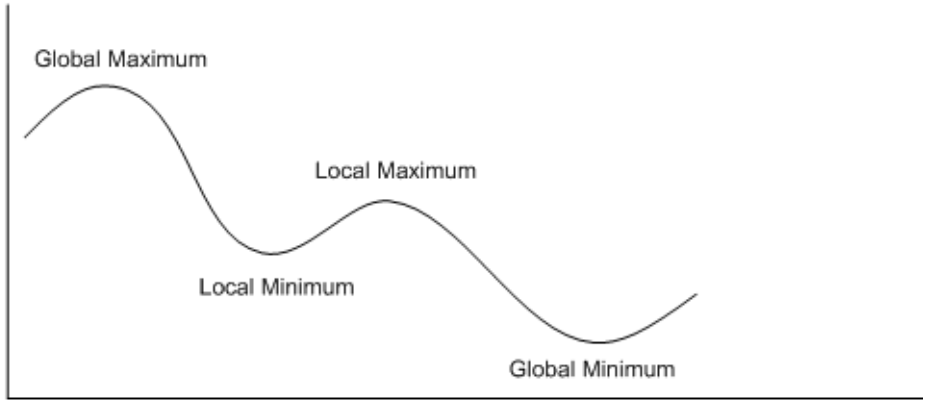
18

Global optimum

Otsingustrateegia „*hill climbing*“

- Sobib juhul, kui
  - on üks sidus olekusiirde diagramm
  - leidub üks optimum, st. kasufunktsioon annab ühese eelistuse
- otsing on lokaalne st põhineb ainult vahetult järgmiste olekute võrdlemisel
- Ei sobi globaakse optimumi leidmiseks, kui on palju lokaalseid optimume

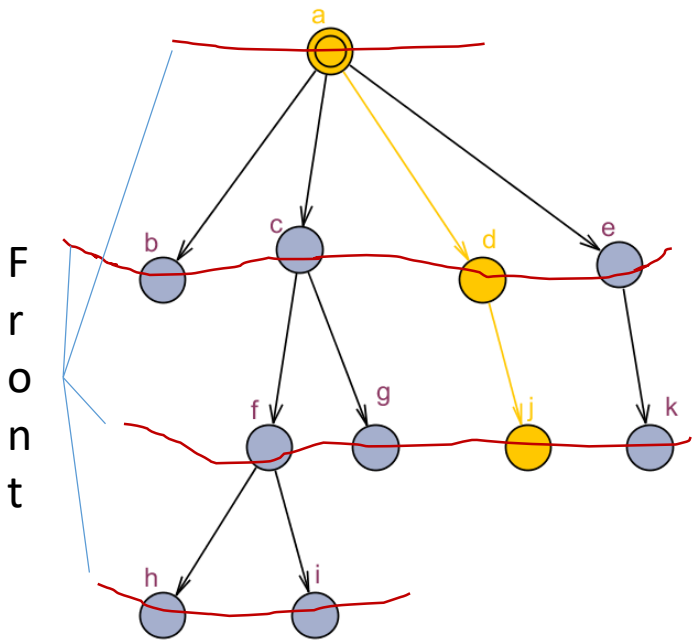




Otsingustrateegia „parim-esmalt“ (*best first*)

- 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 olekus `State`

Näide: parim-esmalt




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

- **state(State,Path,Value)**

```
    % state/3 fakte tekib
 % sama arv, kui otsingu-
      % frondis on tippe
```

„*Parim esmalt*" otsingu kodeerimine

```
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([],S,P,H,F,F).
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(Ms,State,Path,[State1|History],F0,F1).
```

Otsing mängupuul

- Vaatame 2 mängija mängusid (kabe, male, tik-tak, jne)
- mängija käik on siire mängu ühest seisust (olekust) teise
- käikusid tehakse kas korda-mööda või käiguõigus oleneb eelmise käigu tulemusest (n. kabes peale võtmist saab uuesti käia)
- kummalgi mängijal on oma kasufunktsioon, 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), % mängu algseisu genereerimine
        display_game(Position,Player),     % algseisu kuvamine
        play(Position,Player,Game).        % käikude rekursiivne genereerimine

play(Position,Player,Result):-             % Lõpetamistingimuse kontroll
        game_over(Position,Player,Result),!, announce(Result).
play(Position,Player,Result):-
        choose_move(Position,Player,Move), % Käigu valimine
        move(Move,Position,Position1),     % Käigu sooritamine
        display_game(Position1,Player),    % Uue seisu kuvamine
        next_player(Player,Player1),!,     % Järgmise käiguõiguse otsustamine
        play(Position1,Player1,Result).    % Uue käigu genereerimine
```

Mängupuu

- Mängupuu läbimine on sarnane kasufunktsiooniga olekutepuu läbimisele.
- Mängupuud on reeglina liiga suured täieliku otsingu tegemiseks
- Mitte-täieliku otsingu strateegiad:
  - minmax (mängija maksimeerib enda ja minimeerib vastasmängija 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)` `% Move` peab olema antud seisus teostatav käik
- `evaluate_and_choose(Moves,Position,Record,BestMove)` tagastab parima käigu
    - `Record –` 3. argument - antud seisust parima seni leitud käigu skoor, rekursiooni käigus vaadatakse järjest läbi kõik alternatiivid ja parema käigu leidmisel parameetrit uuendatakse.

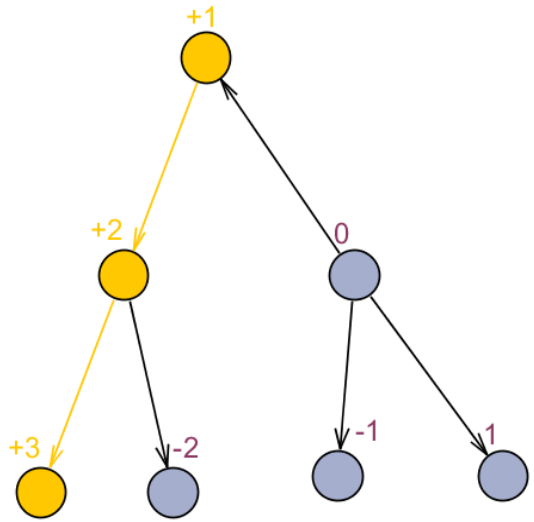
Minmax algoritm

- Mänguseisu kasufunktsiooni arvutamiseks kasutame ettevaatavat planeerimist s.t. arvutame mängu võimaliku seisu mitu sammu ettepoole ja kaugeima parima tulemi põhjal valime jooksva käigu.
- Algoritm eeldab, et vastasmängija teeb niisuguse käigu, mis maksimeerib tema kasufunktsiooni ja minimeerib antud mängija oma.
- olenevalt planeerimise sügavusest (mitu käiku ette vaadatakse)

  leitakse viimase vaadeldava sammu skoori alusel kavandatava käigu

  kasufunktsioon.
- `flag` – muutuja, mis näitab 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),     % minmax planeerimine horisondini D
        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):-        % kui planeerimissügavus saavutatud
        value(Position,V),
        Value is V * MaxMin.
minmax(D, Position, MaxMin, Move, Value):-        % kui planeerimissügavus ei ole saavutatud
        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 teadaolevast parema tulemuseni, siis otsing seda haru pidi lõpetatakse.
- http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html
- Otsingu parameetrid:
  - alfa – vähim garanteeritud ülatõke lahendile (maksimeerivale mängijale)
  - beeta – suurim garanteeritud alamtõke lahendile (minimeerivale mängijale)
- Minimeerissammul peatatakse haru jätkamine, kui väärtus < alfa
- Maksimeerival sammul peatatakse haru jätkamine, kui leitud väärtus > beeta
- Otsingut jätkatakse antud haru pidi, kui $\alpha \leq$ hind $\leq \beta$

Alfa-beeta otsingureegel Prologis

```
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):-     % kui planeerimissügavus saavutatud
        value(Position,Value).
alpha_beta(D,Position1,Alpha,Beta,Move,Value):-    % kui planeerimissügavus ei ole saavutatud
        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):-
      Value =< Alpha.
cutoff(Move,Value,D,Alpha,Beta,Moves,Position,Move1,BestMove):-
      evaluate_and_choose(Moves,Position,D,Value,Beta,Move,BestMove).
```

Kodutöö

Ülesande kirjeldus

- Koostada Prologis kabeprogramm, mis sooritab korraga ühe käigu või võtmise, kui võimalik.
- Programm peab võistlema vastase programmiga.
- Predikaat “arbiiter” annab programmidele korda-mööda käiguõiguse või korduva käiguõiguse, kui eelmine oli võtmine.
- Mäng lõpeb, kui ühel mängijatest ei ole enam võimalik teha käike. Võitja on programm, mis sooritas viimase käigu.
- Arbiiter kontrollib käikude õigsust ja diskvalifitseerib reegleid rikkunud 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).
```

Kabeprogrammi vormistamise reeglid

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

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

- Mooduli peapredikaat peab olema kujul

```
mooduli_peapredikaat(Color).
```

`Color` – nuppude värv (1 või 2), millega antud programm mängib.

- Moodulis ei tohi esineda staatilisi fakte `ruut/3` ja deklaratsiooni `:- dynamic ruut/3.`
- Mooduli peapredikaat ei tohi lõpetada *fail*-ga st. tagurdamine ei tohi minna teise mängija programmi.
- Selleks on soovitav defineerida peapredikaadis lisaks alternatiiv, mis tagastab alati *true*

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

Arbiiteri kohandamine programmidele

- Faktis

```
players_turn(1,2,Nimi1).
```

tuleb 3nda argumendi väärtuseks kirjutada programmi peapredikaadi nimi, mis mängib MUSTADE nuppudega.

- Faktis

```
players_turn(2,1,Nimi2).
```

tuleb 3nda argumendi väärtuseks kirjutada selle programmi peapredikaadi nimi, mis mängib VALGETE nuppudega.

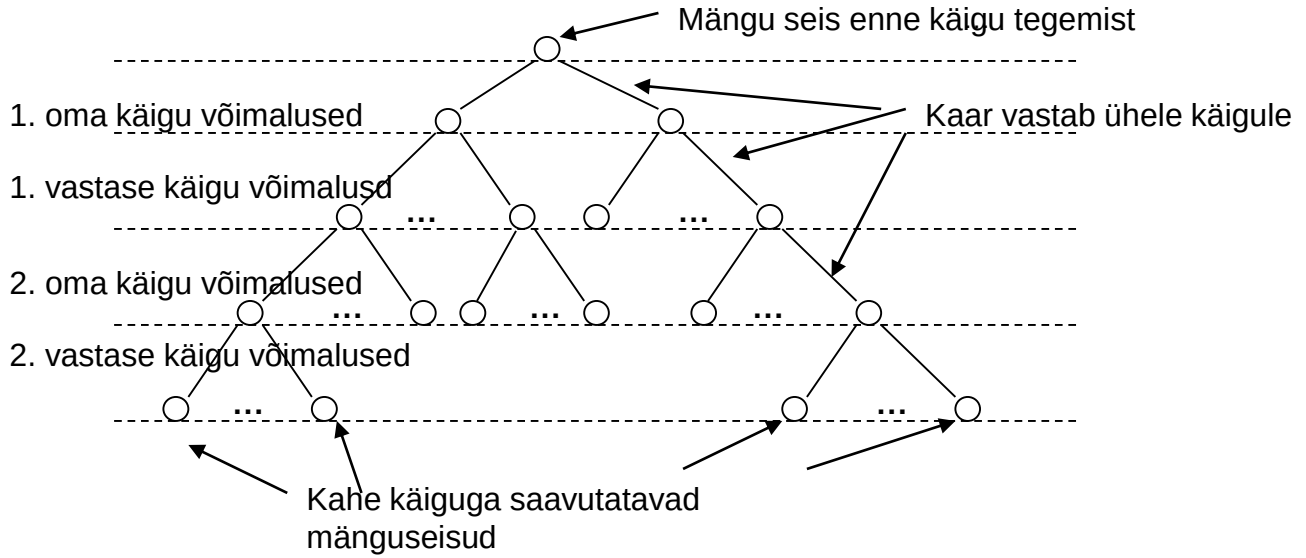
Mängu käivitamine

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

```
?- turniir.
```

Abistavaid näpunäiteid käikude planeerimisel (predikaat `choose_move`)

- 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

- Planeerimispuu implementeerimiseks vajalike faktide loomine,
- Näiteks ruutusid 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 [1,2,10,20]
- `Plan_step` -- planeerimispuu tase, mida antud fakt kirjeldab (vahemikus [0,..,n])
- `Prev_state` -- eelmise seisu ID, millest jõuti antud seisu
- `Present_state` -- planeerimissammu vaadeldava seisu ID
- `Cost` -- seisu kasufunktsiooni väärtus.

  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 pärast n-dal planeerimissammul parima seisu leidmist.
- Parima käigu valimine:
  - Kasutades fakti `ruut/7` parameetri `Cost` väärtusi, leida planeerimispuu terminaalsetele tippudele vastavate (st 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 laual faktide hulgaks `ruut/3` ja anda juhtimine tagasi arbiiterprogrammile.
- Lisaks võib kasutada mängupuu genereerimisel alfa-beeta kärpimist st kärpida planeerimisel harud, mis on kehvemad kui teadaolev parim.