Loeng 10: Otsingustrateegiate programmeerimine

Jüri Vain

ITI0211

Otsing olekusiirde graafidel: **planeerimisülesanded**

- **Planeerimisülesande** kirjeldus peab defineerima otsinguruumi st.
  - Võimalikud otsingu seisundid - olekud,
  - olekutevahelised siirded,
  - alg- ja lõppoleku.

- Planeerimisprobleemi lahendiks on *tee* (olekute jada), mis viib *algolekust* soovitud *lõppolekusse* nii, et teele seatud *lisakitsendused* oleks täidetud.

Otsing olekusiirde graafidel

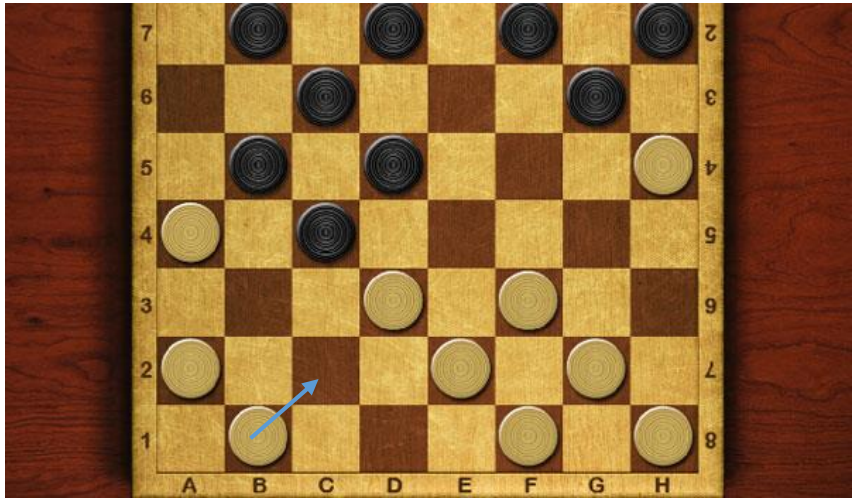
- **Näide 2**: logistika ülesanne

```
% predikaat "state/1" näitab sõiduki asukohta
state(vabaduse_väljak).
…
state(akadeemia_tee).

% predikaat"move/3" esitab transpordiühendusi
move(sõpruse,troll(sõpruse,keemia)).
…
move(estonia,buss(estonia,kaubamaja)).

% predikaat finaal_state/1 esitab sihtolekut
final_state(akadeemia_tee).
```

Otsing olekusiirde graafidel: **planeerimisülesanded**



<u>Näide 1</u> (kabe):

- nuppude asetus 8 x 8 kabelaual on mängu seisu kirjeldav *olek*
  - *N. olekuvektor:* (<$valge_B$,…,$valge_H$>$_1$,…,<$must_A$,…,$vaba_H$>$_8$)
- Nuppudega käimised on *siirded* ühest olekust teise.

Näiteks käik B1 –> C2 kujutub olekuvektoril:

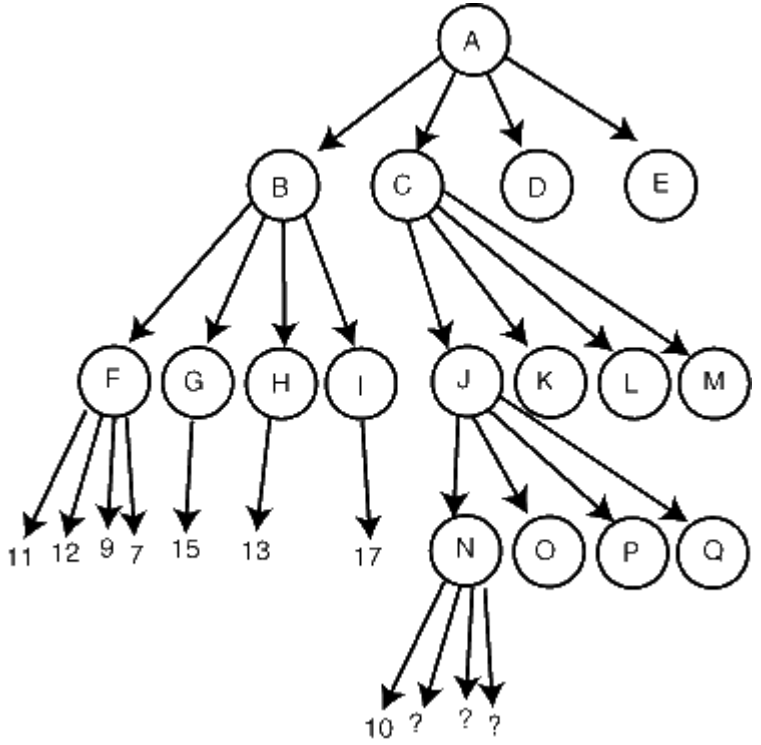
(<$valge_B$,…,$valge_H$>$_1$, <$valge_A$,$vaba_C$,…,$valge_H$>$_2$ …,<$must_A$,…,$vaba_H$>$_8$)

↓

(<$vaba_B$,…,$valge_H$>$_1$, <$valge_A$,$valge_C$,…,$valge_H$>$_2$ …,<$must_A$,…,$vaba_H$>$_8$)

Otsingupuu

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



Sügavuti otsing (dfs) otsingupuul

```
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).  % Päringu näide
Moves = [estonia,vabaduse_väljak,… ]
```

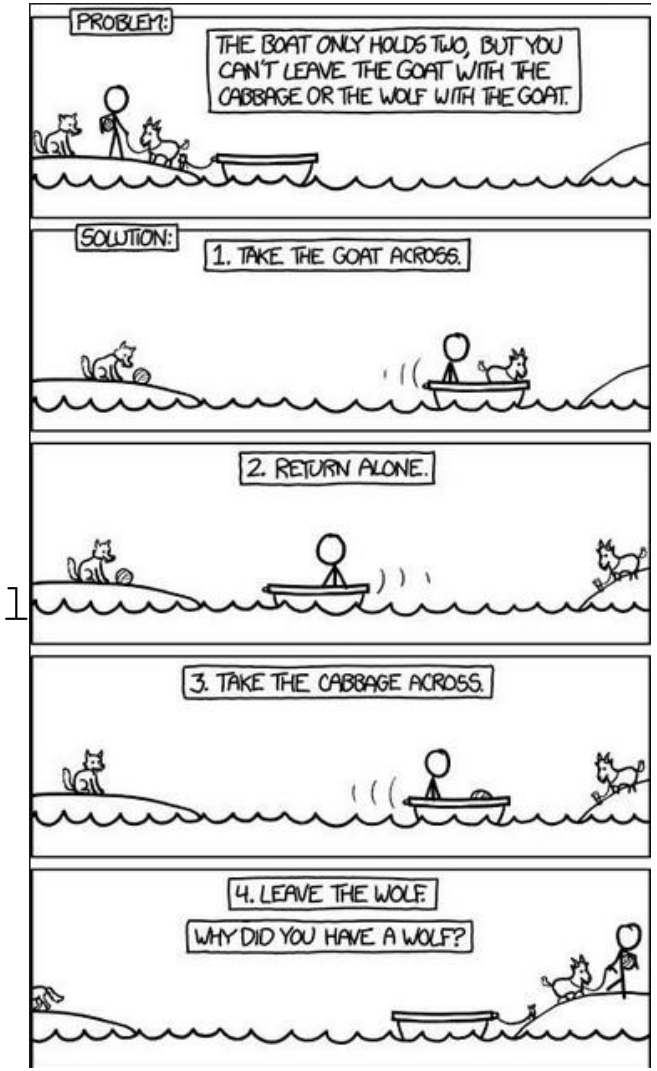
Kuidas defineerida predikaadid state/3, move/2, legal/1?

- Näide:

- Objektid: mees, hunt, kits ja kapsas

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

```
% Boat – paadi asukoht: vasak, parem
% LeftBank – objektide list, mis on vasakul kaldal
% RightBank – objektide 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
```

Jooksev olek — Uus olek

```
update(+state(B,LB,RB), +Cargo, -state(B1,LB1,RB1):-
      update_boat(B,B1),                              % uuenda paadi olekut
      update_banks(Cargo,B,LB,RB,LB1,RB1).            % uuenda kallaste 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).                % Kui mees sõidab üksi, siis kallaste olek ei muutu.
update_banks(Cargo,vasak,L,R,L1,R1):-        % Kui vasakult kaldalt paremale, siis toimub
        select(Cargo,L,L1),                  % elemendi eemaldamine vasaku kalda objektide listist ja
        insert(Cargo,R,R1).                  % elemendi lisamine parema kalda objektide listi.
update_banks(Cargo,parem,L,R,L1,R1):-        % Kui paremalt kaldalt vasakule kaldale, siis toimub
        select(Cargo,R,R1),                  % elemendi eemaldamine parema kalda objektide listist
        insert(Cargo,L,L1).                  % ja elemendi lisamine parema kalda objektide listi.
```

Kuidas defineerida predikaati update/2: abipredikaadid

```
%------------ Elemendi eemaldamine listist -----------------
% Eeldus:  listi elemendid on unikaalsed
select(El,[El|L],L).                    % Kui eemaldatav element on listis välimine element
select(El,[El1|L],[El1|L1]):-           % Kui eemaldatav element ei ole listis  välimine,
       select(El,L,L1).                 % tee rekursiooni samm listi sabaga

%------------ Elemendi lisamine listi ------------------
insert(El,List,List1):-
       sort([El|List],List1).           % Termide sorteerimine termide standardjärjestusse
                                        % Tagastatakse järjestatud list.
```

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

Kuidas esitada kitsendus „kui paat mehega on ühel kaldal, siis teisel kaldal ei tohi olla objektide 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).
```

Näites toodud kitsendavad predikaadid on piisavad, et leida sellele otsinguülesandele lahend.

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 läbivaadatavate käikude hulka kasutades nn *kasufunktsiooni*:

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

  Prologis: `value(State, Value):- ….`

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

Otsingustrateegia „*hill climbing*“

- Sügavuti otsingu (*dfs*) üldistus, kus hargnemisel ei valita otsingupuus vasakult paremale järgmist läbimata haru, vaid haru, millel on suurim kasufunktsiooni väärtus.
- Kasutades üldist *dfs*-reeglit, asendame reegli kehas pöördumise `move(State,Move)` pöördumisega predikaadiga `hill_climb(State,Move)`.
- `hill_climb(State,Move)`:
  - genereerib kõik antud olekust ühe siirdega saavutatavad olekud,
  - järjestab need olekud kasufunktsiooni väärtuse kahanevas järjekorras s.t. tagurdamisel valitakse suurima kasufunktsiooni väärtusega läbimata haru.

Sügavuti otsing *dfs* (slaid 6 uuesti)

```
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).  % Päringu näide
Moves = [estonia,vabaduse_väljak,… ]
```

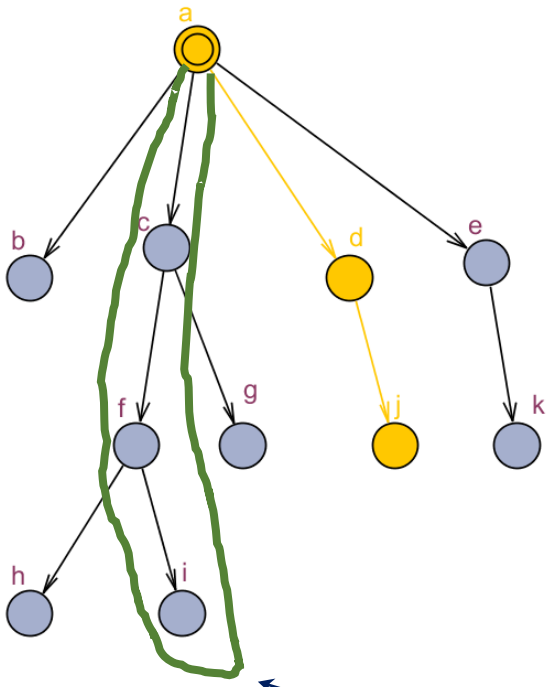
Otsingustrateegia „*hill climbing*“

Nimetame ümber ja kohaldame dfs reeglit:

```
solve_hill_climb(State, History,[]):-                % Kui on jõutud sihtolekusse
        final_state(State).
solve_hill_climb(State, History, Moves):-            % Kui jooksev olek ei ole sihtolek
        hill_climb(+State,+Moves,-Move),             % Predikaadi move asendamine
        update(State,Move,State1),
        legal(State1),
        not member(State1, History),
        solve_hill_climb(State1, [State|History],Moves).
```

```
hill_climb(+State, +Moves, -Move):-
        evaluate_and_order(+Moves, +State,[],-MVs),  % Leia „parim esimene“ järjestus
        member((Move,Value),MVs).                    % Tagurdamisel valitakse paremuselt
                                                     % järgmine siire
```

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).
```

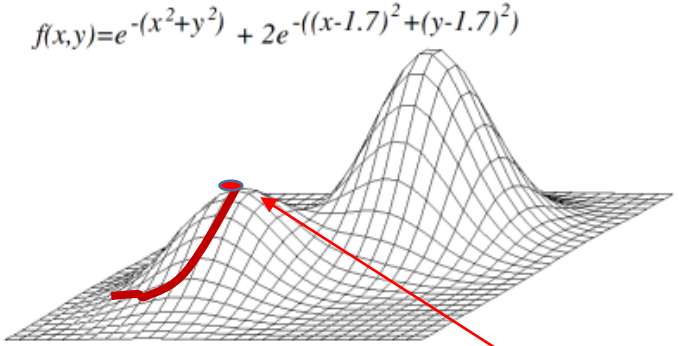
Lokaalne optimum: olek j väärtusega 8:
Haru: <move(a,d), move(d,j) >

18

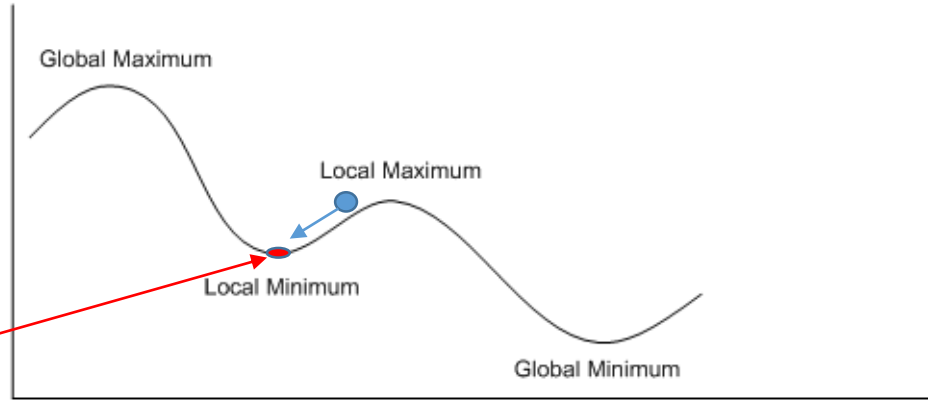
Globaalne optimum

Otsingustrateegia „*hill climbing*“ eelised ja puudused

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



Otsing takerdub lokaalsesse optimumi!



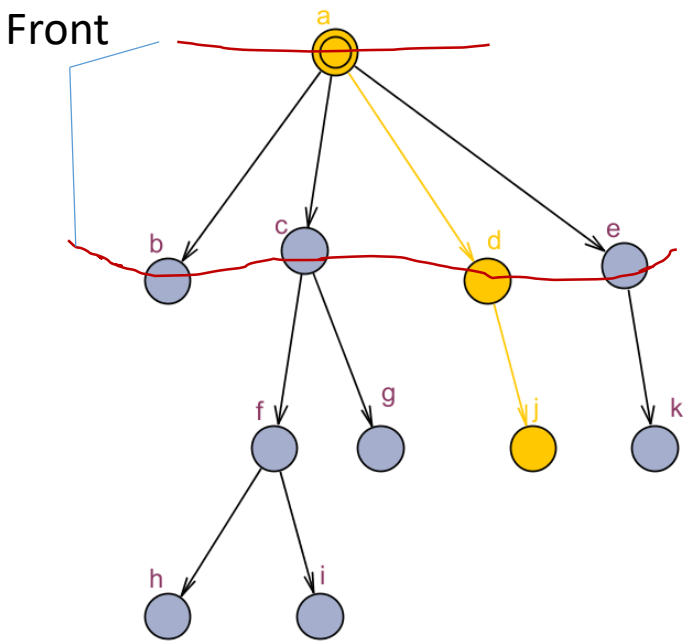
Otsingustrateegia „parim-esmalt“ (*best first*)

- Sarnane *hill climbing’*le.
- Sügavuti otsingu asemel kasutab *laiuti otsingut* (*bfs*) ja *hinnafunktsiooni.*
- Hulk *front* sisaldab vaheolekuid, milleni on otsingu käigus jõutud.
- Järgmise siirde valimisel vaadatakse hulga *front kõiki* olekuid ja nendest lähtuvaid siirdeid.
- Prologis kasutame oleku esitamiseks fakti `state/3`

```
state(State,Path,Value)
```

  - `State` – oleku nimi
  - `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_bestfirst([state(State,Path,Value)|Front], History, BestPath):-
        final_state(State), reverse(Path, BestPath).
solve_bestfirst ([state(State,Path,Value)|Front], History, BestPath):-
        findall(M, move(State, M), Moves),                  % frondi kõikide olekutele leiame
        update_front(Moves,State,Path,History,Front,Front1), % järgnevad olekud
        solve_bestfirst (Front1,[State|History], BestPath).


update_front([],S,P,H,F,F).                          % kui ei ole järgnevaid olekuid
update_front([M|Ms],State,Path,History,F,F1):-       % kui on veel järgnevaid olekuid
         update(State, M, State1),
         legal(State1),
         value(State1,Value),
         not member(State1,History),
         insert((State1,[M|Path],Value),F,F0),
        update_front(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 saab vastase nupu võtmise järel uuesti käia).
- Mõlemal mängijal on oma kasufunktsioon (f1 ja f2), mida ta püüab käigu tulemusena maksimeerida (0-summa mängu korral kehtib f1=-f2).
- Esimese käigu õigus oleneb mängust (esimene käik loositakse, mitme järjestik mängu korral esmakäigu õigus vaheldub, jne).

Näide: mängu peapredikaat

```
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 planeerimine
        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 eelmise näite predikaadis `choose_move/3`

Strateegia rakendamine

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

- `move(.,.)` peab olema antud seisus teostatav käik;
- käigu vastavust reeglitele kontrollib predikaat `legal/1`
- `evaluate_and_choose(Moves,Position,Record,BestMove)` tagastab parima käigu
    - Parameeter `Record` – antud seisust parima seni leitud käigu skoor,
    - Sellega võrreldakse rekursiooni käigus järjest kõik alternatiivid ja parema käigu leidmisel uuendatakse parameetri `Record` väärtust.

Üldistused mänguteoorias: 2-mängija mäng

Kirjeldame mängu ühe mängija seisukohast:

$G = (S, \rightarrow, \text{-->}, s_0, Bad, Goal)$,

kus

- S: mängu seisude hulk
- $\rightarrow \subseteq S \times S$: mängija käigud
- $\text{-->} \subseteq S \times S$: oponendi käigud
- $s_0 \in S$: algseis
- $Bad \subseteq S$: seisude hulk, kus mängija on kaotanud
- $Goal \subseteq S$: seisude hulk, kus mängija on võitnud

Üldistused mänguteoorias: 2-mängija mäng

- Run: is a finite or infinite sequence $r = (s_0, s_1, s_2, \ldots)$ of states
- Safe run: If there is no bad state ($\forall s_i \in r.\ s_i \notin Bad$) in the run.
- Player Strategy: A set of constraints σ that determines which transition a player chooses next being in some state of the game

$$\sigma : S \to S \text{ such that } \forall r \in \rho, r.s \in S : (r.s, \sigma(r.s)) \in E,$$

where ρ is a set of runs, such that from any state s of a run r in ρ, strategy σ chooses the next state reachable by an outgoing edge of s.

Üldistused mänguteoorias: 2-mängija mäng

- Safe strategy: No outgoing transitions in the state ($s \in S$) leads to bad states:

$$\sigma_{safe} = \{(s_i, s_j) \mid s_i, s_j \in S \wedge \ (s_i, s_j) \in E \wedge s_j \notin Bad\}.$$

- Feasible strategy: safe strategy that reaches the goal state ($s \in Goal$) :

$$\sigma_{feasible} = \{(s_i, s_j) \mid (s_i, s_j) \in \sigma_{safe} \wedge \ (\exists s : s \in S \wedge s = s_j \wedge s \in Goal)\}.$$

- Feasible run: A safe run *r of* finite length $|r|$:

$$r_{feasible} = \{r | (\forall s_i : s_i \in r, i \in [1, |r| - 1] : s_i \notin Bad) \wedge (\exists s_j : s_j \in r \wedge \ j = |r| \wedge s_j \in Goal)\}.$$

Üldistused mänguteoorias: 2-mängija mäng

- Optimal run: if it is feasible and reaches the goal state ($s \in Goal$) in the run with a minimum cost:

$$r_{optimal} = \{r | r \in r_{feasible} \wedge cost(r) \leq min(ran(cost))\}.$$

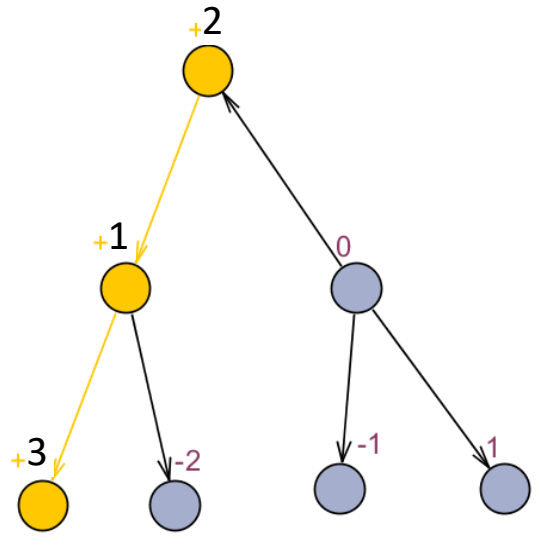
where cost function $cost : \rho \rightarrow \mathbf{R}$ assigns a real-valued number to each run.

- Winning (optimal) strategy: if it is a safe and feasible strategy and there is a run ending up in goal state (*Goal*) with a minimum cost:

$$\sigma_{optimal} = \{(s_i, s_j) \,|(s_i, s_j) \in \sigma_{feasible} \wedge \ (\exists r : r \in r_{optimal} \wedge s_i \in r \wedge \ s_j \in r)\}.$$

Minmax algoritm

- Mänguseisu kasufunktsiooni arvutamiseks kasutame **ettevaatavat planeerimist** s.t. arvutame mängu võimaliku seisu n sammu ettepoole ja n sammu järgse parima tulemuse põhjal valime jooksva käigu.
- Algoritm eeldab, et vastasmängija teeb niisuguse käigu, mis maksimeerib tema kasufunktsiooni ja minimeerib antud mängija oma.
- `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(Position, Move, Position1),
        minmax(D,Position1,MaxMin,MoveX,Value),      % minmax planeerimine sügavuseni D
        update(Move,Value,Record,Record1),           % MaxMin -  kasufunktsiooni väärtus
        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
        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 (parameetris Record), 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 kasufunktsiooni väärtus < alfa
- Maksimeerival sammul peatatakse haru jätkamine, kui leitud kasufunktsiooni väärtus > beeta
- Kui $\alpha \leq$ kasufunktsiooni väärtus $\leq \beta$, siis otsing antud haru pidi jätkub.

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,Position,Alpha,Beta,Move,Value):-      % kui planeerimissügavus ei ole saavutatud
        findall(M,move(Position,M),Moves),
        Alpha1 is -Beta,                % olenevalt kelle käiku planeerimisel vaadatakse, vahetub alfa ja beeta
        Beta1 is -Alpha,                % väärtus ning omandab vastupidise märgi
        D1 is D-1,                      % tegemata planeerimissammude väärtus kahaneb
        evaluate_and_choose(Moves,Position,D1,Alpha1,Beta1,nil,(Move,Value)).
```

Kärpimine

```
cutoff(Move,Value1,D,Alpha,Beta,Moves,Position,-(Move,Value)):-
      Value >= Beta.          % maksimeerimissammu peatumistingimus
cutoff(Move,Value,D,Alpha,Beta,Moves,Position, BestMove):-
      Value =< Alpha.         % minimeerimissammu peatumistingimus
cutoff(Move,Value,D,Alpha,Beta,Moves,Position, BestMove):-
      evaluate_and_choose(Moves,Position,D,Value,Beta,Move,-BestMove).
```

Kodutöö

Ülesande kirjeldus

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

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,MustadegaMängivProgr).
```

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

- Faktis

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

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

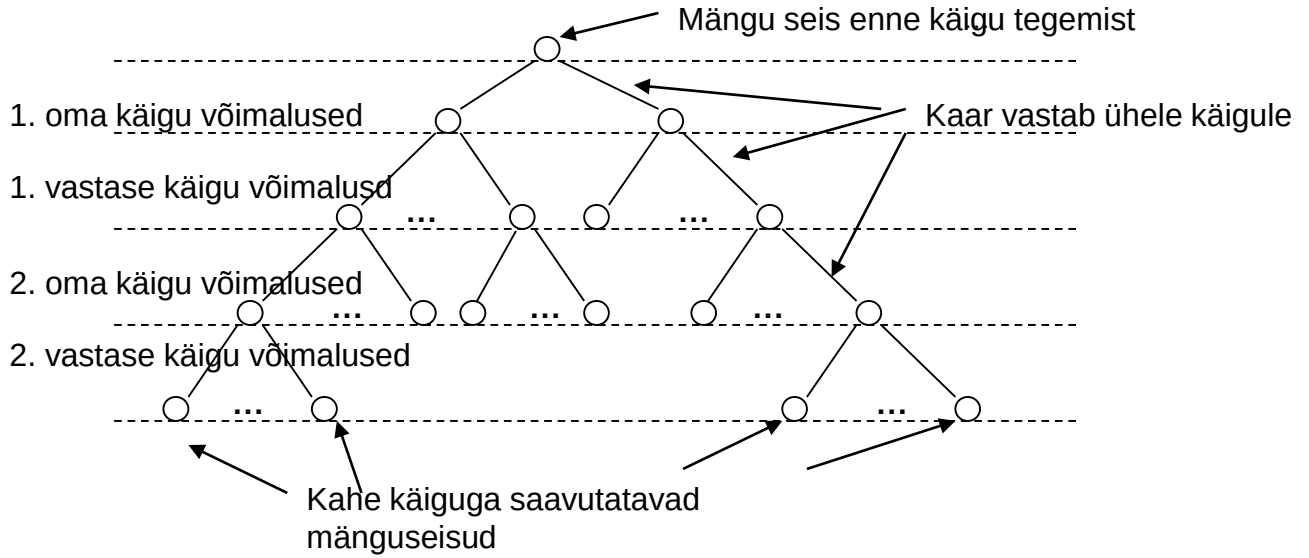
Mängu käivitamine

- Faktide `players_turn` 3ndate parameetrite seadistamine programmis `arbiter.pl`
- Laadida mällu programm `arbiter.pl`
- Laadida mällu mängijate programmid
- Käivitada mäng päringuga

```
?- 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 kabe ruutusid kirjeldav 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 käesoleva 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 nuppude asetus 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.

- Head nuputamist!