Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

RAČUNALNIŠKO IGRANJE IGER S KARTAMI

DIPLOMSKA NALOGA
Univerzitetni študij

Mitja Luštrek
Mentor prof. dr. Ivan Bratko

5. julija 2002

KAZALO

I. POVZETEK

Računalniško igranje iger s popolno informacijo je dokaj dobro raziskano, igranje iger z nepopolno informacijo (med katere sodijo tudi igre s kartami) pa dosti manj. V prvem delu je pregled metod, ki se dajo uporabiti za računalniško igranje iger s kartami. Pri tem je poudarjena bolj praktična plat (algoritmi) kot teoretična (izsledki iz teorije iger), saj je namen naloge uporabiti zbrane informacije za izdelavo programa za igranje taroka. Najprej so opisani algoritmi za preiskovanje drevesa igre, ki se sicer uporabljajo predvsem v igrah s popolno informacijo: poglavitni je alfa-beta, ki se da na mnoge načine izboljšati. Nato je razloženo, kako se lahko uporabijo tudi v igrah z nepopolno informacijo: tu je glavna metoda vzorčenje Monte Carlo. Predstavljen pa je še primer nekoliko drugačne metode – planiranja. Za konec je opisanih nekaj konkretnih programov za igranje iger s kartami: za poker in bridž (ki sta predmet tako akademskih raziskav kot komercialnih programov) ter za tarok (za katerega je najti le ljubiteljske izdelke).

Drugi del je posvečen programu za igranje taroka, ki sem ga izdelal, da bi splošne metode za igranje iger s kartami preizkusil v praksi. Najprej so opisana pravila taroka za tri igralce, kakršnega igra moj izdelek. Nato so predstavljeni uporabljeni algoritmi. Jedro programa je močno nadgrajena različica preiskovalnega algoritma alfa-beta, ki se uporablja skupaj z vzorčenjem Monte Carlo. Opisanih pa je še nekaj postopkov, ki se nanašajo prav na tarok. Razloženo je, kako in kje je uporabljeno človeško znanje o taroku, kajti izkaže se, da je samo preiskovanje drevesa igre za učinkovit program premalo. Opisano je tudi, kako deluje program z igralčevega vidika. Nato pa je podana še ocena mojega izdelka: meritve hitrosti algoritma in koristi, ki jih prinašajo posamični elementi, ter pogled človeških igralcev nanj. To dvoje kaže, da je uporabljeni preiskovalni algoritem učinkovit in da program solidno igra tarok.

2. UVOD

Igranje iger je že zgodaj pritegnilo raziskovalce s področja umetne inteligence. Claude Shannon [1], Alan Turing [2] in morda še posebej Arthur Samuel s svojim programom za igranje dame [3] so se tega področja lotili že v 50. letih 20. stoletja. Ob pomoči bliskovitega razvoja strojne opreme pa dandanes programi za igranje iger dosegajo in presegajo največje mojstre med ljudmi. Najodmevnejši dosežek te vrste je nedvomno zmaga, ki jo je Deep Blue slavil nad svetovnim prvakom v šahu Garijem Kasparovom. Podobno dobri so tudi programi za *backgammon*, pri dami, *Othellu* in *Scrabblu* so od ljudi sploh boljši, blizu so jim tudi pri bridžu in pokru, nekaterim igram – npr. goju – pa računalniki še niso kos. [4]

Večina raziskav računalniškega igranja iger se ukvarja z igrami s popolno informacijo (predvsem s šahom), igre z nepopolno informacijo (kakršnih je denimo večina iger s kartami) pa so nekoliko zapostavljene. Eden izmed razlogov je bržkone ta, da je šah verjetno najbolj dognana, organizirana in ugledna igra, tako da se kar ponuja kot izbira za razvoj računalniških programov. Šahovski algoritmi pa so z več ali manj prilagoditvami potlej uporabni še za množico drugih iger s popolno informacijo. Nič manj pomemben razlog pa najbrž ni, da so igre z nepopolno informacijo trši oreh, kar je bilo še posebej problematično v preteklosti, ko računalniki niso bili tako zmogljivi kot danes.

Je pa res, da je osnovna in najpogostejša metoda – to je preiskovanje drevesa igre – uporabna pri obeh kategorijah. Pri igrah z nepopolno informacijo se navadno uporablja v kombinaciji s kako vrsto simulacije: to pomeni, da tvorimo nabore manjkajočih podatkov in iščemo po vsakem izmed njih. Uporabljajo pa se tudi povsem drugačne metode. Metode za preiskovanje drevesa igre opišem v podpoglavju 3.1. V podpoglavju 3.2 se nekoliko posvetim igram z nepopolno informacijo nasploh, nato pa opišem, kako je pri njih moč uporabiti preiskovanje drevesa igre, kakšne probleme to prinese in kako jih rešujemo. V podpoglavju 3.3 predstavim še primer nekoliko drugačne metode – planiranja.

Sam sem se osredotočil na igre s kartami ravno zato, ker so slabše raziskane, za tarok, ki mi je osebno zelo všeč in sem se mu zato še posebej posvetil, pa nisem zasledil, da bi se z njim sploh kdo resneje ukvarjal. Nekaj več pozornosti uživata le poker in bridž. Razlog je najbrž ta, da sta precej razširjena in organizirana, zato sta tudi dokaj dobro raziskana. To pa seveda lajša delo vsem, ki se lotevajo programov za njuno igranje. Da ne omenjam, da takšni programi laže najdejo kupce. V podpoglavju 3.4 opišem, kar mi je uspelo izbrskati o programih za igranje pokra in bridža, omenim pa še dva programa za tarok – ne toliko zato, ker bi premogla kake posebne kvalitete, kot zato, ker sta po moji vednosti sploh edina svoje vrste in povrh še slovenska izdelka.

Zbrane podatke o računalniškem igranju iger s kartami sem preizkusil na programu za igranje taroka, ki sem ga poimenoval Silicijasti tarokist in je internetski javnosti na voljo na strani **http://tarok.bocosoft.com**. Z njim je moč igrati tarok za tri igralce (pravila so opisana v podpoglavju 4.1, napotki za uporabo pa so v podpoglavju 4.3).

Pri svojem delu sem se oprl na tehnike, razvite za igranje bridža, ker je bridž od iger, o katerih se da najti kaj koristnih informacij, še najbolj podoben taroku. Najprej sem si za zgled vzel GIB (opisan v razdelku 3.4.2), ker je videti, da je vrhunski primerek svoje vrste, in ker se najdejo članki o njegovem delovanju. GIB z vzorčenjem Monte Carlo (opisanim v razdelku 3.2.2) naključno tvori primere nasprotnikovih kart. Nato na njih uporabi različico preiskovanja drevesa igre, imenovano particijsko iskanje (opisano v razdelku 3.1.3), ki deluje tako, da za vsako vozlišče poišče množico takih, ki jim pripada enaka ocena, in nato to množice skupaj z

njeno oceno shrani v tabelo. Kadar naslednjič pride v vozlišče, ki spada v eno izmed shranjenih množic, njegovo vrednost le prebere iz tabele.

Vzorčenje Monte Carlo je dokaj preprosta ideja, zato ga ni bilo pretežko implementirati. GIB sicer uporablja nekoliko nadgrajeno različico (opisano v razdelku 3.2.3), vendar igranja ne izboljša bistveno, poleg tega pa je ni prav enostavno sprogramirati. Pri močno dodelanem programu se kljub temu splača potruditi, saj tudi majhna izboljšava ni zanemarljiva, moj pa še ni tako zrel, zato sem ostal pri osnovni različici.

Particijsko iskanje se je izkazalo za trši oreh in mi ga ni uspelo implementirati. Eden izmed razlogov je prav gotovo, da mi njegove podrobnosti niso znane (kar ni presenetljivo, glede na to, da je GIB komercialen program), poleg tega pa razvoj GIBa traja dosti dlje kot razvoj Silicijastega tarokista. In kajpada moram dopustiti, da je njegov avtor Matthew L. Ginsberg sposobnejši od mene. Tako sem od particijskega iskanja ohranil le osnovno idejo: da v tabelo shranjujem množice vozlišč (v nasprotju z bolj običajno, da se v tabelo shranjujejo posamična vozlišča). Particijsko iskanje dejansko izračuna, katera vozlišča imajo enako vrednost, jaz pa jih v skupine združujem hevristično. Poleg tega sem uporabil še druge naprednejše metode za preiskovanje drevesa igre, ki so opisane v podpoglavju 3.1. Podrobnosti o uporabljenih algoritmih so v podpoglavju 4.2.

Opravil sem tudi nekaj meritev hitrosti delovanja mojega programa; predvsem sem si prizadeval ugotoviti, koliko posamične izboljšave algoritma alfa-beta prispevajo k hitrosti. Izkazalo se je, da je pri globini iskanja, kakršna se uporablja pri igranju, prihranek časa mojega iskanja v primerjavi z golim algoritmom alfa-beta 86-kraten. Vsi izsledki so zbrani v razdelku 4.4.1.

Program sem preizkusil tudi s človeškimi igralci. Tu sicer nisem podal prav objektivne ocene, ker bi bilo za kaj takega potrebno dolgotrajno preizkušanje številnimi tarokisti. Sem pa opravil preizkus s pomočjo dveh dobrih igralcev ter zapisal njuni mnenji in naša skupna opažanja o programovi igri (v razdelku 4.4.2).

Ker po moji vednosti ne obstaja prav dosti slovenske literature o računalniškem igranju iger, sem bil primoran poiskati precej prevodov angleških izrazov. Pri njih sem v oklepaju z *ležečo pisavo* zapisal izvirni izraz.

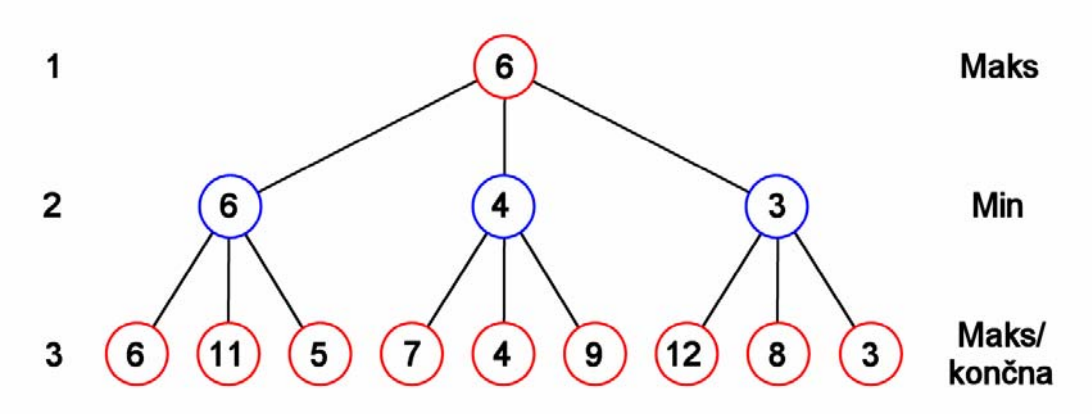
3. SPLOŠNO O IGRANJU IGER S KARTAMI

3.I. PREISKOVANJE DREVESA IGRE

3.I.I. OSNOVA – MINIMAKS IN ALFA-BETA

Igranje igre lahko predstavimo kot drevo, v katerem so vozlišča stanja igre, povezave pa poteze. Izmenjujejo se plasti, kjer smo na potezi mi, in plasti, kjer je na potezi nasprotnik. Tu naj pripomnim, da je običajnejši izraz nivo, vendar se pri igranju iger navadno uporablja plast (*ply*). [7] Če bi tako drevo razvili do konca, bi lahko natančno predvideli potek igre in vedno izbrali najboljšo potezo (v nekaterih igrah bi lahko celo vedno izbrali potezo, po kateri bi vsi nasprotnikovi odgovori pripeljali v stanje, kjer lahko zmagamo).

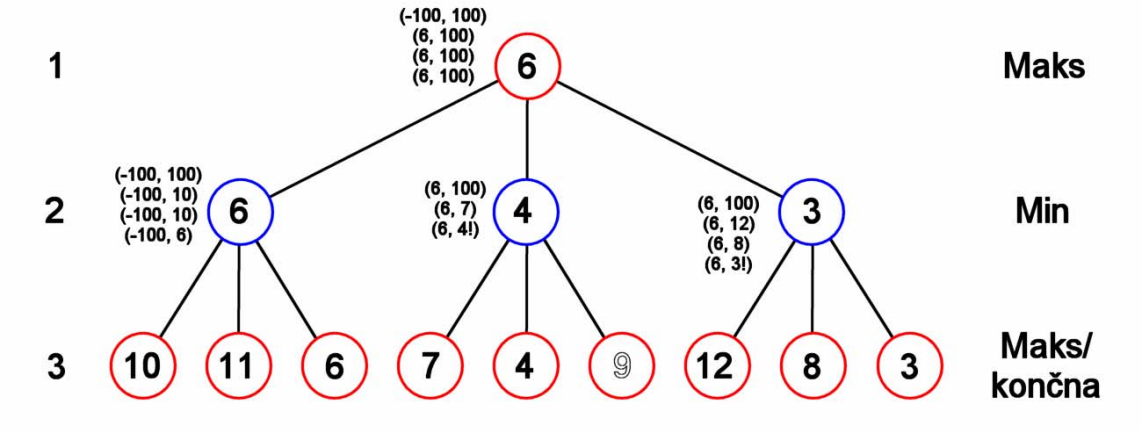
Ker je v večini iger celotno drevo preveliko, da bi ga razvili, uporabimo algoritem minimaks [5]. Po njem drevo razvijemo do izbrane globine, nato pa vsak list ocenimo. Ocenjevalna funkcija je odvisna od problema, s katerim se ukvarjamo. Recimo, da si prizadevamo za čim večji rezultat: vozlišča *maks* so potem tista, kjer smo na potezi mi, kajti tam izberemo vejo z največjim rezultatom; v vozliščih *min* je na potezi nasprotnik in v njih izberemo vejo z najmanjšim rezultatom. To ponazarja slika 1. Izraza *maks* in *min* pogosto uporabljamo tudi zase in za nasprotnika, ne le za vrste vozlišč.



Slika 1: Drevo igre pri algoritmu minimaks

Očitno pa se v algoritmu minimaks opravi nekaj odvečnega preiskovanja. V primeru na sliki 1 vzemimo, da drevo razvijamo od leve proti desni. V tem primeru bi v vozlišču 4 v plasti 2 lahko *min* pri listu 4 iskanje prekinil, saj bi že vedel, da se v tem vozlišču da doseči vrednost vsaj 4; ker pa je v prejšnjem vozlišču v plasti 2 dosegel vrednost 6, bo *maks* v plasti 1 raje izbral to vozlišče – temu rečemo rez. Algoritem, ki izkorišča to lastnost, se imenuje alfa-beta [6] in je osnova večine današnjih programov za igranje iger – to je algoritem 1.

Ime alfa-beta izvira iz spremenljivk α in β, ki označujeta zgornjo in spodnjo mejo vrednosti, ki jo trenutno lahko dosežemo. Kako bi ga uporabili na primeru s slike 1, kaže slika 2: pri vozliščih so dodani pari (α, β), mesta, kjer pride do reza, pa so označena s klicajem.



Slika 2: Drevo igre pri algoritmu alfa-beta

```
Alfa-beta (vozlišče, alfa, beta)
  če je vozlišče list
    vrni njegovo vrednost
  sicer
    a := alfa
    b := beta
    če je vozlišče vrste min
      za vse naslednike vozlišča
        vrednost := Alfa-beta (trenutni naslednik vozlišča, a, b)
        b := min (b, vrednost)
        če b <= a
          vrni b
      vrni b
    sicer
      za vse naslednike vozlišča
        vrednost := Alfa-beta (trenutni naslednik vozlišča, a, b)
        a := max (a, vrednost)
        če a >= b
          vrni a
      vrni a
```

Algoritem 1: Alfa-beta

3.1.2. TRANSPOZICIJSKA TABELA

Pogosto se dogaja, da se v drevesu večkrat pojavi isto stanje igre – npr. če dve bolj ali manj neodvisni potezi naredimo v različnem vrstnem redu, obakrat pridemo do istega stanja. Če si prvič to stanje zapomnimo, nam naslednjič ni treba preiskovati drevesa pod njim, ampak zgolj preberemo njegovo vrednost iz tabele. [7] Ta tabela se imenuje transpozicijska (*transposition table*) in je navadno implementirana kot zgoščena tabela. Shranjeno stanje lahko uporabimo, kadar je bila globina drevesa pod njim večja ali enaka kot globina drevesa, ki ga moramo trenutno še preiskati. Če je takrat, ko smo stanje shranili, prišlo do reza, shranjene vrednosti ne moremo uporabiti kot točno vrednost, ker zaradi reza ne vemo, kakšna točna vrednost je, lahko

pa ga uporabimo za prilagoditev vrednosti spremenljivke α ali β. V tem primeru je pametno tudi najprej uporabiti potezo, ki je prej povzročila rez, ker je verjetno, da ga bo spet. V algoritmu 2 so deli, ki se nanašajo na transpozicijsko tabelo, označeni z *ležečo pisavo*.

```
Alfa-beta-trans (vozlišče, alfa, beta)
   če je vozlišče list
     vrni njegovo vrednost
   sicer
     a := alfa
     b := beta
     če je vozlišče v transpozicijski tabeli
       če je vrste
         točno: vrni vrednosti iz tabele
         spodnja meja: a := vrednost iz tabele
         zgornja meja: b := vrednost iz tabele
     če je vozlišče vrste min
       za vse naslednike vozlišča (začenši z onim iz tabele)
         vrednost := Alfa-beta-trans (trenutni naslednik vozlišča, a, b)
         b := min (b, vrednost)
         če b <= a
           prekini zanko
       rezultat :=  b
     sicer
       za vse naslednike vozlišča (začenši z onim iz tabele)
         vrednost := Alfa-beta-trans (trenutni naslednik vozlišča, a, b)
         a := max (a, vrednost)
         če a >= b
           prekini zanko
       rezultat := a
     vrsta := točno
     če vrednost >= b
       vrsta := spodnja meja
     če vrednost <= a
       vrsta := zgornja meja
     shrani v transpozicijsko tabelo (vozlišče, rezultat, vrsta)
     vrni rezultat
```

Algoritem 2: Alfa-beta s transpozicijsko tabelo

3.1.3. PARTICIJSKO ISKANJE

Mnogo stanj, ki so shranjena v transpozicijski tabeli, si je zelo podobnih, zato bi jih bilo zaželeno združiti. Tudi npr. pri taroku nam intuicija pravi, da navadno ni važno, ali držimo srčevo dvojko ali trojko.

To izkorišča particijsko iskanje (*partition search*) [8]. Za njegovo razlago je potrebnih nekaj definicij.

Naj bo S množica stanj igre.
Funkcija s (p) vrne množico naslednikov stanja p.
R_0 (S) je množica stanj, iz katerih se da doseči S: množica p-jev, za katere velja s (p) ∩ S ≠ Ø.
C_0 (S) je množica stanj, iz katerih je neizogibno doseči S: množica p-jev, za katere velja s (p) ⊆ S.

Particijski sistem (*partition system*) igre sestavljajo tri funkcije, ki vračajo množice stanj igre. Za te množice velja, da ocenjevalna funkcija vsem njihovim elementom priredi enako vrednost.

P (p) vrne stanja, ki so dovolj podobna p, da so ocenjena enako.
R (p, S) vrne stanja, ki se dajo doseči iz S, in mora vsebovati p; R (p, S) $\subseteq R_0$ (S), kar pomeni, da je R konzervativen približek za R_0 (uvajamo ga zato, ker utegne biti pretežavno točno izračunavati R_0).
Podobno C (p, S) vrne stanja, ki jih je neizogibno doseči iz S, in mora vsebovati p.

```
Alfa-beta-part (vozlišče, alfa, beta)
   če je (S, alfa, beta, vrednost), vozlišče ∈ S v transpozicijski tabeli
     vrni (vrednost, S)
   sicer če je vozlišče list
     vrni (vrednost vozlišča, P (vozlišče))
   sicer
     S_vse := ∅
     če je vozlišče vrste min
       v_rezultat := beta
       za vse naslednike vozlišča
         (v_nova, S_nova) := Alfa-beta-part (trenutni naslednik vozlišča,
           alfa, v_rezultat)
         če v_nova <= alfa
           shrani v transpozicijsko tabelo (S_nova, alfa, beta, v_nova)
           vrni (v_nova, S_nova)
         če v_nova < v_rezultat
           (v_rezultat, S_rezultat) := (v_nova, S_nova)
         S_vse := S_vse ∩ S_nova
       če v_rezultat = 1
         S_rezultat := C (vozlišče, S_vse)
       sicer
         S_rezultat := R (vozlišče, S_rezultat) ∩ C (vozlišče, S_vse)
     sicer
       v_rezultat := alfa
       za vse naslednike vozlišča
         (v_nova, S_nova) := Alfa-beta-part (trenutni naslednik vozlišča,
           v_rezultat, beta)
         če v_nova >= beta
           shrani v transpozicijsko tabelo (S_nova, alfa, beta, v_nova)
           vrni (v_nova, S_nova)
         če v_nova > v_rezultat
           (v_rezultat, S_rezultat) := (v_nova, S_nova)
         S_vse := S_vse ∩ S_nova
       če v_rezultat = 0
         S_rezultat := C (vozlišče, S_vse)
       sicer
         S_rezultat := R (vozlišče, S_rezultat) ∩ C (vozlišče, S_vse)
   shrani v transpozicijsko tabelo (S_rezultat, alfa, beta, v_rezultat)
   vrni (v_rezultat, S_rezultat)
```

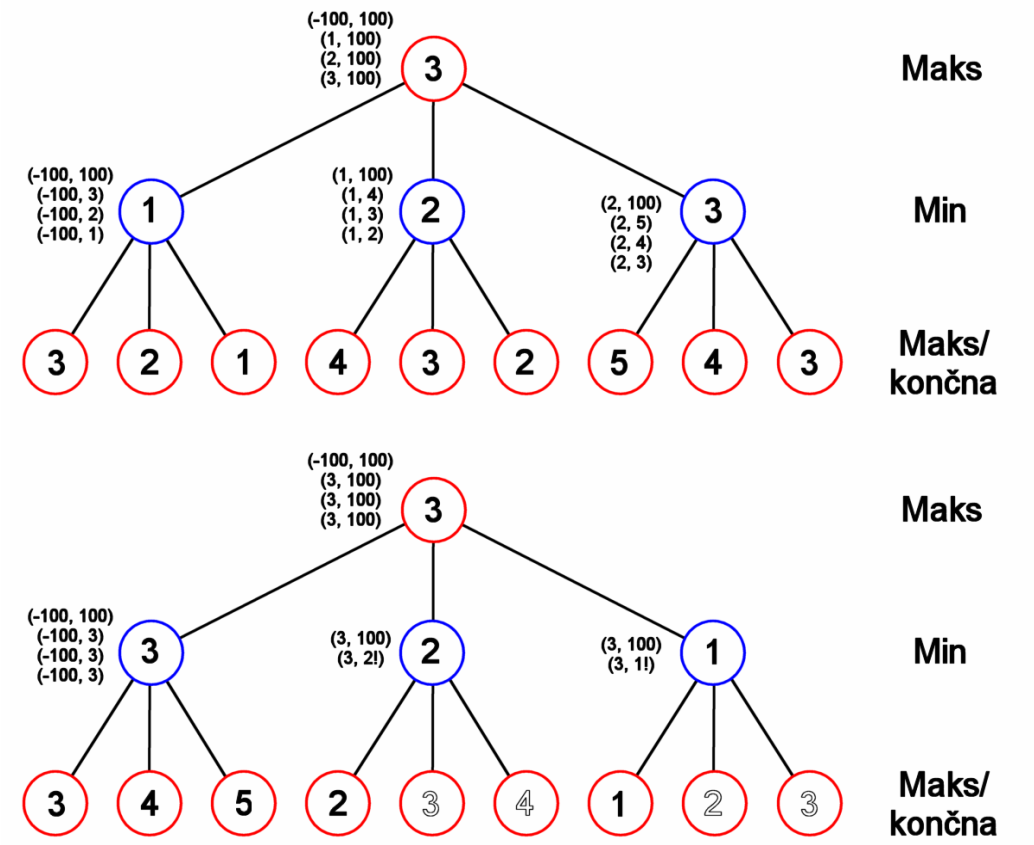
Algoritem 3: Particijsko iskanje

Algoritem 3 kaže particijsko iskanje. Prikladno je imeti vrednosti vozlišč na intervalu [0, 1], kjer 1 pomeni zmago, 0 pa poraz. Dela, ki sta vredna posebne pozornosti, sta označena z *ležečo pisavo*. Če vsa vozlišča naslednje plasti vodijo v zmago (v vozliščih *min*) ali poraz (v vozliščih *maks*), se kot rezultat vrne množica vseh stanj, iz katerih je neizogibno doseči eno izmed stanj naslednje plasti. Očitno jih namreč lahko prav tako obravnavamo kot dobljene oziroma izgubljene. Sicer pa se kot rezultat vrne množica stanj, iz katerih je možno doseči najugodnejše stanje naslednje plasti – R (vozlišče, $S_{rezultat}$), obenem pa je iz njih neizogibno doseči vsaj kako

stanje naslednje plasti – C (vozlišče, S_{vse}). Prvi pogoj nam zagotavlja, da imamo na voljo možnost, ki smo jo izbrali v konkretnem primeru, drugi pa, da nimamo na voljo kake boljše možnosti, ki je v konkretnem primeru nismo upoštevali.

3.1.4. RAZVRŠČANJE POTEZ

Iskanje se močno pospeši, če v vsakem vozlišču najprej preiščemo najboljšega naslednika. Meja iskanja se namreč v tem primeru že po prvem nasledniku nastavi na končno vrednost za trenutno vozlišče in pri neprvih vozliščih hitro pride do rezov. Slika 3 kaže koristnost razvrščanja – vozlišča v zgornjem drevesu so razvrščena slabo (rezov sploh ni), v spodnjem pa dobro (rezi so povsod, kjer je možno).



Slika 3: Razvrščanje potez pri algoritmu alfa-beta

Najenostavnejši način za razvrščanje potez je iterativno poglabljanje [4]. Najprej se drevo preišče eno plast v globino. Vozlišča se razvrstijo na podlagi rezultatov te plasti, nakar se preišče dve plasti v globino. Tako se rezultati vsake iteracije uporabijo za usmerjanje naslednje.

Že omenjen način je transpozicijska tabela. Če podatki o stanju igre niso zadostni za uporabo vrednosti iz tabele, se lahko najprej uporabi poteza, ki se je izkazala za najboljšo prejšnjič, ko smo bili v enakem stanju. Ker utegne biti pri nekaterih iskanjih transpozicijska tabela premajhna, tako da se veliko položajev prepiše, se uporabljaja tudi ovržbena tabela (*refutation table*) [7]. Ta je manjša in shranjuje le poteze, ki so v preteklosti povzročile reze.

Podobna metoda je raba ubijalske hevristike (*killer heuristic*) [9]. V različnih vozliščih iste plasti drevesa se pogosto ista poteza izkaže za najboljšo (ubijalsko). Tako je za vsako plast

smotrno shraniti nekaj potez (navadno eno ali dve), ki jih nato preizkusimo najprej, če so v trenutnem vozlišču seveda veljavne.

To lahko posplošimo v zgodovinsko hevristiko (*history heuristic*) [10]. Vsaki potezi priredimo neko vrednost in kadarkoli v vozlišču ugotovimo, da je poteza najboljša, ji to vrednost povečamo. Za koliko jo povečamo, je navadno odvisno od tega, kako globoko smo preiskali drevo od trenutnega vozlišča navzdol – če smo ga daleč, je bolj zanesljivo, da je poteza dobra, zato jo je takrat smiselno povečati bolj.

In seveda so nam vedno na voljo metode, ki temeljijo na znanju o problemu, s katerim se ukvarjamo.

3.1.5. PRILAGAJANJE ŠIRINE ISKALNEGA OKNA

Algoritem alfa-beta v osnovni različici začne iskanje z oknom $(-\infty, \infty)$. Pogostost rezov pa se da zvečati, če sta ti dve vrednosti bliže pričakovani najboljši vrednosti. Če pričakujemo rezultate okrog n, iščemo z oknom $(n - \delta,\ n + \delta)$. Ta metoda se imenuje aspiracijsko iskanje (*aspiration search*) [4]. Izid iskanja bo eden izmed tehle:

$n - \delta <$ rezultat $< n + \delta$: iskanje je bilo krajše, kot bi bilo z navadnim algortimom alfa-beta;
rezultat $\leq n - \delta$: iskanje je treba ponoviti z oknom $(-\infty,$ rezultat);
rezultat $\geq n + \delta$: iskanje je treba ponoviti z oknom (rezultat, ∞).

Aspiracijsko iskanje se pogosto rabi skupaj z iterativnim poglabljanjem – rezultat prejšnje iteracije se lahko uporabi kot pričakovani rezultat trenutne.

Širina iskalnega okna se prilagaja že pri osnovnem iskanju alfa-beta. Če je vrednost prvega naslednika trenutnega vozlišča n, bomo pri drugem uporabili okno (n, β). To velja, če je trenutno vozlišče vrste *maks* – če je vrste *min*, se ravna podobno. Če uporabljamo razvrščanje potez, je pričakovati, da bodo vrednosti neprvih naslednikov manjše in bo zato pri njih hitro prišlo do reza. Takrat lahko poizkusimo zgolj pokazati, da so neprvi nasledniki slabši, ker je to ceneje kot polno preiskovanje. To storimo z iskanjem z najmanjšim oknom (*minimal windows search*, tudi *principal variation search*) [7]. Vsa neprva vozlišča preiščemo z oknom $(n, n + 1)$ – z najmanjšim oknom. Če kje dobimo rezultat, večji od n, moramo pri tem nasledniku iskanje ponoviti z oknom (rezultat, β).

3.1.6. PRILAGAJANJE GLOBINE ISKANJA

Čeprav algoritem alfa-beta načeloma išče do vnaprej določene globine, se zdi smotrno bolj zanimive veje preiskati globlje od manj zanimivih. [4] Vendar tovrstne metode niso tako zelo razširjene kot druge izboljšave algoritma alfa-beta, pa tudi povečini so odvisne od problema, s katerim se ukvarjamo. Ena izmed takih metod je iskanje z upoštevanjem stabilnih stanj (*quiescence search*) [9]. Stabilna vozlišča so taka, od katerih ne vodi dosti zanimivih možnosti, zato se lahko ocenijo z malo nadaljnjega iskanja. Tovrstno iskanje se pogosto uporablja pri šahu.

3.1.7. ALTERNATIVE

Algoritem alfa-beta in njegove izpeljanke imajo skoraj popoln monopol med preiskovalnimi algoritmi, ki se uporabljajo za igranje iger. A nekaj alternativ [4] vendar obstaja.

B* denimo poišče optimistično in pesimistično vrednost vsakega lista. Te vrednosti se nato prenašajo navzgor po drevesu in iskanje teče, dokler v korenu ne najdemo poteze, katere pesimistična ocena je vsaj tako dobra kot optimistične ocene ostalih. Za to potezo tako vemo, da je najboljša.

Zarotniški (*conspiracy numbers*) algoritem beleži število listov, katerih vrednost se mora spremeniti (ki se morajo zarotiti), da se spremeni koren.

Obstajajo pa tudi drugi algoritmi, a kakor ta dva so slabo raziskani in v praksi še niso dokazali svoje vrednosti.

3.2. OBRAVNAVANJE NEPOPOLNE INFORMACIJE

3.2.1. SPLOŠNO O IGRAH Z NEPOPOLNO INFORMACIJO

Nepopolno informacijo v igrah si lahko predstavljamo kot npr. nabor različnih možnih razporeditev kart med igralce (recimo temu množica svetov), igralci pa ne vedo, katera je prava (v katerem svetu so). [11] Drevo igre z nepopolno informacijo je podobno drevesu igre s popolno informacijo, le da imajo listi po več vrednosti (za vsak svet svojo). Seveda je možno, da v nekaterih svetovih nekateri listi niso dosegljivi – te v tistih svetovih pač zanemarimo. Za tako predstavitev pa mora biti izpolnjen vsaj pogoj, da se v vseh svetovih na enak način izmenjujejo plasti *min* in *maks*.

Najti optimalno strategijo za igro z nepopolno informacijo je NP-poln problem. [12] Njegova časovna zahtevnost je namreč eksponentna glede na velikost drevesa igre (pri minimaksu pa je linearna). Iz tega sledi, da zanimive igre lahko rešimo le približno. To pa bi najbrž v glavnem držalo tudi, če problem ne bi bil NP-poln, kajti pri večini iger s popolno informacijo prav tako ne moremo preiskati vsega drevesa igre (ker je časovna zahtevnost takega iskanje še vedno eksponentna glede na globino drevesa).

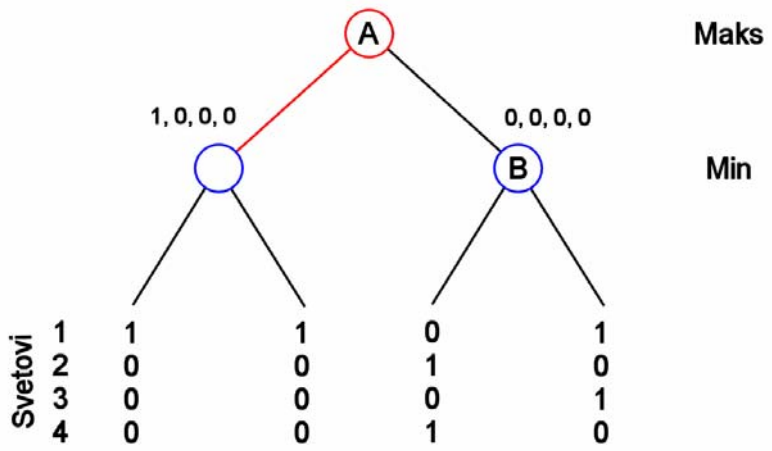
3.2.2. VZORČENJE MONTE CARLO

Vzorčenje Monte Carlo (*Monte Carlo sampling*) [11] naključno izbere nekaj svetov in preišče drevo igre zanje. Načeloma bi lahko preiskali vse svetove, a to je praviloma nepraktično, ker jih je preveč. Če najdemo najboljšo rešitev v dovolj svetovih, je statistično verjetno, da je najpogostejša tudi zares najboljša. Če npr. v 80% primerov ugotovimo, da je najbolje igrati križevega kralja, ga tudi zares igramo. Funkcija za vrednotenje potez je takšna:

$$f(\text{poteza}_i) = \sum_{j=1}^{n} P(s_j) \times \text{vrednost}_{ij} \qquad (1)$$

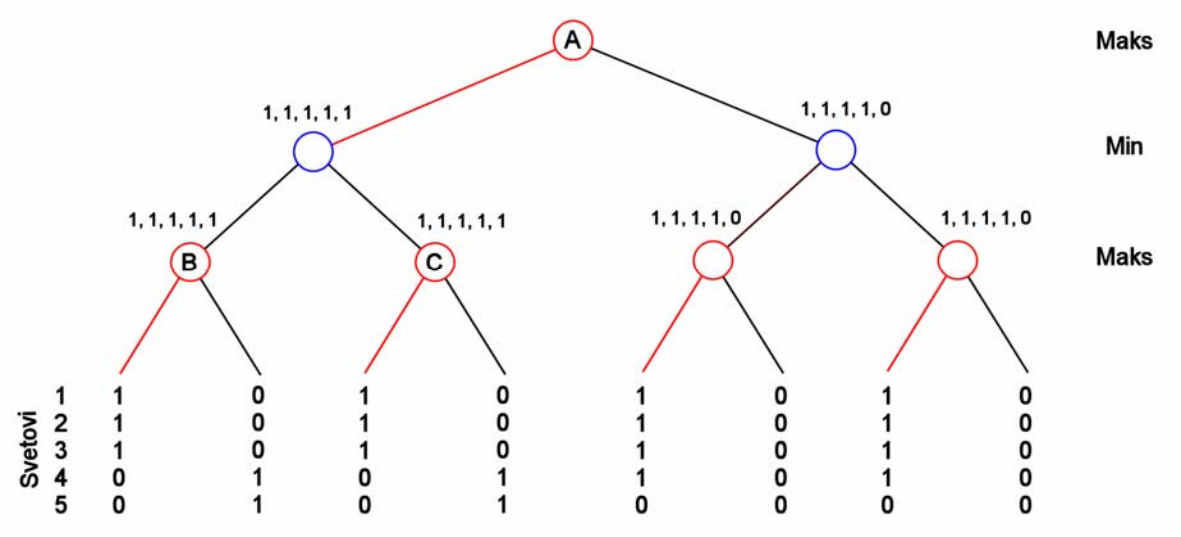
V enačbi (1) je P (s_j) verjetnost, da je izmed n svetov pravi j-ti, $vrednost_{ij}$ pa za i-to potezo in j-ti svet izračunamo z algoritmom minimaks ali kako njegovo izvedenko. P (s_j) je navadno enaka 1. Če ni, je to selektivno vzorčenje (*selective sampling*) [4], ki da enako dobre rezultate pri manjšem številu vzorcev (če seveda izberemo reprezentativne svetove), a ga je teže implementirati.

Vzorčenje Monte Carlo pa ima nekaj težav. Prva je, da **predpostavlja, da ima *min* popolno informacijo o igri**, zaradi česar ne izkoristi dejstva, da je ponavadi nima. Je pa res, da je ta predpostavka dokaj tradicionalna – ne le pri računalniškem igranju iger, ampak tudi npr. pri študiju bridža. Poleg tega pravzaprav ne vemo, koliko nasprotnik ve, zato je morda bolje biti pesimističen. To težavo prikaže slika 4. Na njej *maks* v vozlišču A izbere levo stran, ker je vektor vrednosti po svetovih (na slikah 4-8 označen kot niz števil pri notranjem vozlišču ali stolpec števil pri listu) pri levem poddrevesu ugodnejši (zmaga v četrtini primerov, ne desni pa nikoli). A v resnici to utegne biti napaka, kajti če *min* ne ve, kateri svet je pravi, sta zanj v vozlišču B leva in desna veja enakovredni, tako da bo v polovici primerov izbral napačno in bo *maks* zmagal.



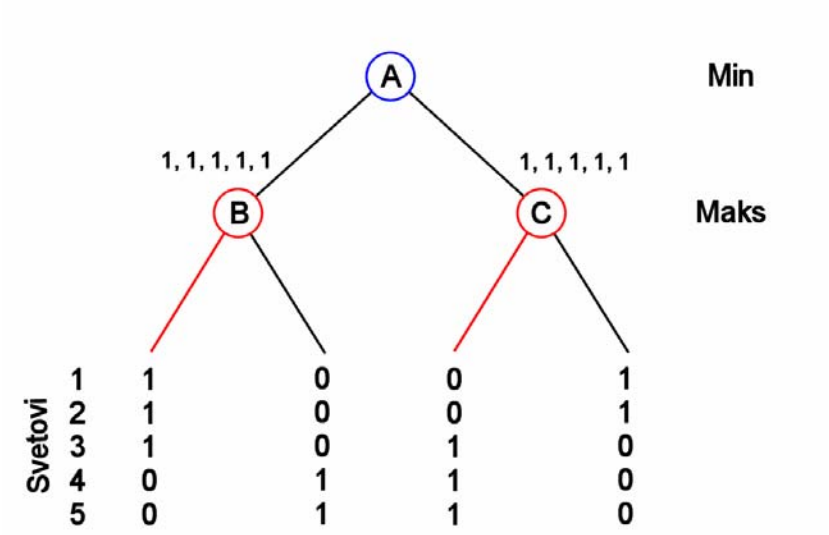
Slika 4: Maks napačno predpostavlja, da je min popolno informiran

Druga težava je, da vzorčenje Monte Carlo **nekatere odločitve prelaga na kasnejše poteze**. [13] Npr. izbiramo med dvema potezama: A in B. Z A zmagamo, če ima križevega kralja prvi nasprotnik in bo naša naslednja poteza C ali pa če ima križevega kralja drugi nasprotnik in bo naša naslednja poteza D. Z B pa zmagamo ne glede na to, kdo ima križevega kralja, razen če ima vse srčeve karte prvi nasprotnik (kar je malo verjetno). Vzorčenje Monte Carlo bo kot pravilno potezo izbralo A, ker z njo lahko zmagamo v vsakem primeru. Vendar to velja ob predpostavki, da bomo do naslednje poteze vedeli, kdo ima križevega kralja, kar pa se najbrž ne bo zgodilo, tako da bi bilo pametneje igrati B. Slika 5 nam prikaže tak položaj. Na njej *maks* v vozlišču A izbere levo stran, ker je vektor vrednosti po svetovih pri levem poddrevesu ugodnejši (zmaga v vseh primerih, ne desni pa le v štirih od petih). A v resnici bo *maks* na levi zmagal le v treh primerih od petih, ker v vozliščih B in C še vedno ne bo vedel, kateri svet je pravi, in bo zato izbral levo vejo.



Slika 5: Maks napačno predpostavlja, da bo še izvedel, kateri svet je pravi

Tretja težava pa je ta, da se pri vzorčenje Monte Carlo ravna, kot da bi bi bilo odločanje v vsakem vozlišču odvisno le od drevesa pod njim – **obnaša se preveč lokalno**. V resnici namreč *min*, če ve, kateri svet je pravi (kar je naša predpostavka), lahko izkoristi maksovo nepopolno informiranost. Seveda pa mora za to *min* vedeti, kako *maks* deluje in koliko v resnici ve, kar sploh ni nujno res. Kaj se utegne zgoditi, če je, nam kaže slika 6. Ker *min* lahko predvidi, kako bo izbral *maks*, bo v vozlišču A v svetovih 1 in 2 izbral desno vejo, v svetovih 4 in 5 pa levo. *Maks* bo tako zmagal le v svetu 3. Če bi *maks* v vozlišču B izbral desno vejo, bi zmagal v dveh primerih (v svetovih 4 in 5). Podobno bi bilo tudi, če bi v vozlišču C izbral desno vejo – zmagal bi v svetovih 1 in 2.



Slika 6: Min izkoristi maksovo neinformiranost

3.2.3. REŠEVANJE PRELAGANJA ODLOČITEV

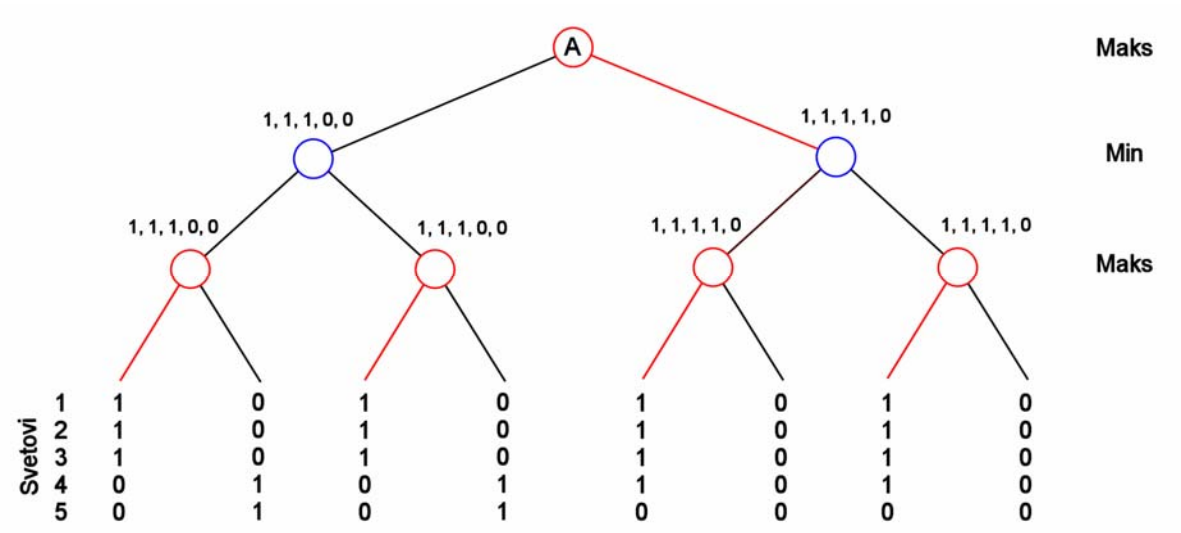
Vse tri naštete težave z vzorčenjem Monte Carlo bi bilo koristno rešiti. A zakaj prva in zadnja nista tako problematični (oziroma ju prav dobro sploh ne moremo rešiti), sem že omenil. Ne gre pa zanemarjati druge, ki utegne povzročiti prenekatero slabo potezo.

Pomagamo si tako, da postavimo predpostavko o pravem svetu in potem igramo, kot da ta predpostavka drži. [13] V ta namen uvedemo dosegljive množice (*achievable sets*). Dosegljiva množica je množica svetov, za katero imamo načrt, ki zmaga v vsakem njenem elementu. Izbrana množica pa mora biti čim bolj verjetna.

Najprej tvorimo vzorec Monte Carlo. Nato njegove elemente enega za drugim poizkusimo dodati v množico. Vsakič preverimo, ali je nova množica še vedno dosegljiva. Če je, element zares dodamo, sicer pa ne. Zaželeno je tudi, da bi bila izbrana dosegljiva množica čim večja (kar pomeni čim bolj verjetna). To lahko dosežemo z optimizacijo s škripajočim kolesom (*squeaky wheel optimization*). Elemente vzorca po vrsti poizkušamo uvrstiti v dosegljivo množico. Pri tvorjenju množice izvedemo več iteracij, elemente, ki nam jih v neki iteraciji ne uspe vključiti (škripljejo), pa pomaknemo na začetek vrste. To povzroči, da v naslednji iteraciji njihova vključitev laže uspe, ker smo, ko pridemo do njih, manj omejeni z že dodanimi elementi.

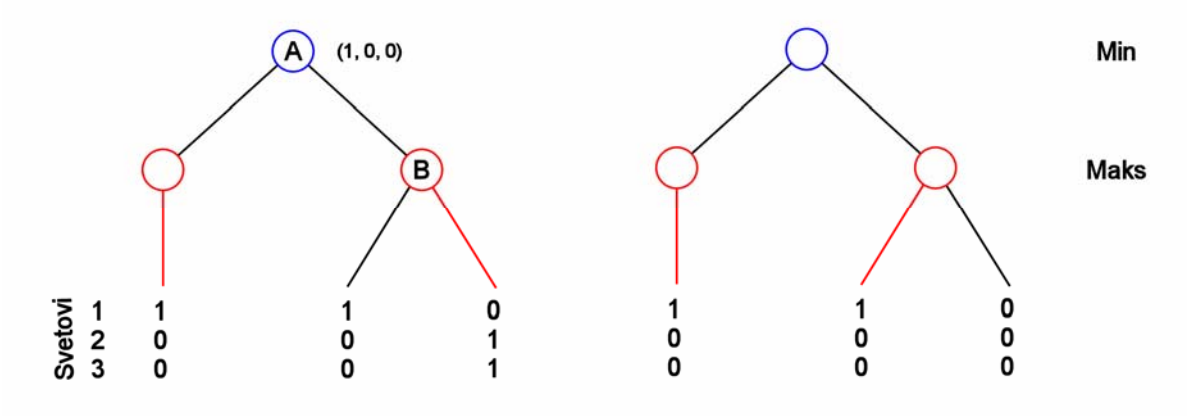
3.2.4. DRUGE IZBOLJŠAVE VZORČENJA MONTE CARLO

Metoda, ki je pravzaprav podobna metodi z dosegljivimi množicami, je **vektorski minimaks** (*vector minimaxing*) [11]. V praksi sicer ni potrjena, a je bolj nazorna od prej opisane in zato vseeno vredna omembe. Od vzorčenja Monte Carlo se razlikuje po tem, da v vektorje vrednosti po svetovih za vsak svet zapišemo vrednost, ki bi jo dosegli z na koncu dejansko izbrano odločitvijo, ne pa da za vsak svet uporabimo svojo odločitev. Slika 7 kaže, da tako rešimo težavo s slike 5 – *maks* se v vozlišču A zdaj pravilno odloči za desno vejo. Metoda z dosegljivimi množicami pa najprej z grobim preiskovanjem drevesa igre določi množico svetov, za katere se da najti rešitev, ki bo pravilna v vseh, nato pa s temeljitim preiskovanjem to rešitev zares poišče.



Slika 7: Vektorski minimaks

Minimaks z zmanjševanjem vrednosti (*payoff-reduction minimaxing*) [11] rešuje tretjo težavo vzorčenja Monte Carlo – njegovo preveliko lokalnost. Na levi strani slike 8 vidimo, da bi se *maks* v vozlišču B odločil za levo vejo. *Min*, ki ima popolno informacijo in se zaveda, kako deluje *maks*, bi zato v vozlišču A izbral desno vejo le v svetu 1, v svetovih 2 in 3 pa bi izbral levo vejo. *Maks* bi tako v vsakem primeru izgubil, čeprav bi mu leva veja v vozlišču B zagotovila zmago vsaj v svetu 1. To lahko rešimo tako, da izvedemo minimaks za vsak svet in si zapomnimo vrednosti, ki jih lahko dosežemo v vozliščih *min* (na sliki v oklepajih). Nato za vsak list poiščemo prvo vozlišče *min* nad njim. V listu vrednost za vsak svet nadomestimo z



Slika 8: Minimaks z zmanjševanjem vrednosti

ustrezno vrednostjo, shranjeno v vozlišču *min*, če je slednja manjša od tiste v listu. Tako popravljene vrednosti nam kaže desna stran slike 8, kjer *maks* zdaj igra pravilno.

Ta algoritem se da še nekoliko izboljšati, če za njegovo osnovo namesto minimaksa vzamemo alfa-beta – nastali algoritem je **beta-zmanjševanje** (*beta-reduction*) [11]. Funkciji min in max definiramo takole:

$$\min_i \vec{K}_i = (\min_i K_i[1], ..., \min_i K_i[n])$$
$$\max_i \vec{K}_i = (\max_i K_i[1], ..., \max_i K_i[n]) \qquad (2)$$

V enačbah (2) je K_i vektor vrednosti po svetovih za i-to poddrevo vozlišča.

Povečanje učinkovitosti ni tolikšno kot pri običajnem algoritmu alfa-beta, pokaže pa se zvečanje pravilnosti. Napake zaradi prevelike lokalnosti nastanejo, kadar se v vozliščih *maks* odločamo med njegovimi poddrevesi, ne da bi upoštevali ostala poddrevesa. Ker je pri beta-zmanjševanju takega odločanja manj, je tudi napak manj.

Metode, opisane v tem razdelku, niso nujno povezane z vzorčenjem Monte Carlo. Če z njimi obdelamo vse svetove, vzorčenja sploh ne uporabimo. Če pa obdelamo naključno izbrano podmnožico vseh svetov, lahko rečemo, da so nadgradnje vzorčenja Monte Carlo. Pokazano je, da z vsemi tremi metodami zvečamo pravilnost igranja (tako pri naključno tvorjenih drevesih igre kot tudi pri primerih iz bridža), niso pa dovolj učinkovite za praktično igranje, tako da so za zdaj zanimive le teoretično.

3.3. PLANIRANJE

Igranje iger s pomočjo preiskovanja drevesa igre je težavno, ker je pri bolj zapletenih in/ali obsežnih igrah to drevo zelo veliko. Problem je še hujši pri igrah z nepopolno informacijo, ker

moramo ne le izbirati med množico potez, ki so na voljo glede na stanje igre, ampak tudi upoštevati različna možna stanja. To rešimo tako, da namesto potez obravnavamo taktike, ki jih je bistveno manj. [14]

Pri tem si lahko pomagamo s planiranjem s hierarhično mrežo nalog (*hierarchical task network planning*) [15]. Začnemo z mrežo nalog, ki predstavljajo stvari, ki jih moramo postoriti. Naloge imajo seznam argumentov (ki so lahko spremenljivke ali konstante) in seznam omejitev, ki jim mora biti zadoščeno, da jih lahko opravimo, ter so lahko razgradljive ali nerazgradljive. Za razgradljive so določene tudi metode, s katerimi jih opravimo. Planiranje poteka tako, da izberemo razgradljivo nalogo, nato poiščemo metodo za njeno razgradnjo in jo nadomestimo z mrežo podnalog, ki jih izbrana metoda predpisuje. Omejitve te mreže podnalog pa vključimo v plan. To počnemo toliko časa, dokler ni cel plan sestavljen iz nerazgradljivih nalog. Če lahko vse spremenljivke pri nalogah opredelimo tako, da je zadoščeno vsem omejitvam, je plan uspešen.

Planiranje s hierarhično mrežo nalog za igranje iger uporabimo tako, da različne taktike predstavimo kot metode, znane podatke o igri kot množice podatkov o stanju (*state information sets*), neznane podatke pa kot verjetnostne funkcije (*belief functions*). [16] Drevo igre zgradimo tako, da v vsakem stanju igre uporabimo vse metode, ki jih lahko (katerih predpogojem zadosti množica podatkov o trenutnem stanju), te nam dajo poteze, ki vodijo do novih stanj, itd. Stanja, kjer smo na potezi mi, ovrednotimo z rezultatom najboljše poteze (tako kot pri minimaksu). V stanjih, kjer je na potezi nasprotnik, pa uporabimo vsoto naslednikov, obteženo z njihovimi verjetnostmi (ki nam jih da verjetnostna funkcija). Tako drevo igre je mnogo manjše od običajnega, tako da ga lahko preiščemo do konca.

3.4. KONKRETNE IGRE

3.4.1. POKER

Kar nekaj ljudi je že preučevalo poker in tudi računalniški programi za njegovo igranje so bili napisani, a malo je bilo dobrih in tudi prav dosti pozornosti niso pritegnili. Izjema je morda le Orac [4], ki ga je sprogramiral poklicni pokeraš Mike Caro in je zmagal v kar nekaj igrah proti vrhunskim igralcem. Žal ni bil nikdar dobro dokumentiran in dovolj temeljito preizkušen. Prva resneje zastavljena programa za igranje pokra sta tako Loki in njegov naslednik Poki [17], ki sta zmožna precej dobrega igranja različice *Texas Hold'em* (ki se igra tudi na svetovnem prvenstvu v pokru), čeravno najboljših med ljudmi (še) ne dosegata.

Texas Hold'em se igre tako, da vsak igralec dobi dve karti (ki ju nasprotniki ne vidijo), nato se stavi, potem vsak igralec dobi tri karte (ki jih vidijo vsi), spet sledijo stave, za njimi dobi vsak še eno karto (ki jo vidijo vsi), spet se stavi in na koncu vsak dobi še sedmo karto (ki jo vidijo vsi), kateri sledi poslednji krog stav. Nato se karte razkrijejo in tisti z najmočnejšimi zmaga.

Poki uporablja verjetnostne trojke (o; i; z), ki pomenijo verjetnost, da je smotrno v naslednjem krogu stav odstopiti, izenačiti stavo ali zvišati. Med tem trojim nato izbere naključno, da se izogne predvidljivosti – npr. pri trojki (0; 0,8; 0,2) ne bi nikoli odstopil, izenačil bi v 80% primerov, zvišal po v 20% primerov.

Kako staviti po prvih dveh kartah, odloči ekspertni sistem na podlagi vnaprej tvorjenih tabel za vse možne pare kart. Nadaljnje stave pa se določijo s simulacijo. S selektivnim vzorčenjem program izbere nekaj najverjetnejših parov skritih kart nasprotnikov, nato pa za vsakega odigra

igro enkrat tako, da izenači, in enkrat tako, da zviša, ter primerja dobiček. Če obe možnosti prineseta izgubo, pomeni, da je najbolje odstopiti. Pri simuliranju nasprotnikovih odločitev tudi uporablja verjetnostne trojke, le da so izračunane na podlagi formule. Ta primerja karte, ki jih ima igralec, s kartami, ki jih lahko imajo njegovi nasprotniki, upošteva pa tudi, kako se utegne stanje spremeniti s kartami, ki jih bodo igralci še dobili.

Ker niso vsi pari skritih kart enako verjetni (npr. igralec, ki ima slabe karte, navadno kmalu odstopi, tako da ima tisti, ki ni odstopil, najbrž dobre karte), se za vsakega nasprotnika vodi tabela uteži, v kateri je vsakemu paru skritih kart pripisana neka verjetnost. Kadar se del kart razkrije (in nekateri pari postanejo nemogoči) in po vsakem krogu stav (ko nasprotnikovo stavljenje da informacije o njegovih kartah) se te uteži prilagodijo. Če je npr. za par asov verjetnostna trojka (0; 0,2; 0,8) in je nasprotnik izenačil, je nova verjetnost enaka prejšnji, pomnoženi z 0,2: stava, ki je v nasprotju s pričakovano, je verjetnost za par asov močno znižala. [18] Na teh verjetnostih temelji selektivno vzorčenje.

Ker različni igralci igrajo različno, Poki modelira tudi sloge posameznih nasprotnikov. Pri tem si med drugim pomaga z nevronskimi mrežami. Zdi pa se, da ta del še ni prav dodelan, ker se v starejših člankih na to temo sploh ne omenja, pa tudi v novejših o njem ni najti dosti podrobnosti.

3.4.2. BRIDŽ

Bridž je igra za štiri igralce, ki sodelujejo po dva in dva. Najprej igralci licitirajo, koliko vzetkov bo kateri par pobral in karte katere barve (če sploh katere) bodo aduti. Zmagovalec licitacije nato igra s svojimi in partnerjevimi kartami (slednje so vidne vsem igralcem). Na izigrano barvo je treba odgovoriti z isto barvo; če je igralec nima, lahko odvrže aduta, če nima niti tega, pa karkoli. Višje karte in aduti poberejo. Par zmaga, če izpolni, kar je izlicitiral.

Računalniško igranje bridža je dokaj priljubljeno in obstaja kar nekaj dobrih programov. A večina jih ne izhaja iz akademskih logov, poleg tega pa so komercialni, tako da o njih ni znanega prav dosti. Izjemi sta Bridge Baron [19] in GIB [20], a kar sem našel podatkov o prvem, so najbrž zastareli, saj se nanašajo na različico 8 iz leta 1997, današnja pa nosi številko 12.

Bridge Baron 8 je za igranje uporabljal planiranje, kot ga opisujem v podpoglavju 3.3. Vdelanih je imel čez 400 različnih metod. Na svetovnem prvenstvu v računalniškem bridžu leta 1997, ki se ga je udeležilo pet tekmovalcv, je zmagal in je v tistem času veljal za najboljši program za igranje bridža.

GIB je bolje dokumentiran, čeravno prav o podrobnostih tudi ni dosti znanega. Za licitiranje uporablja knjižnico z okrog 3000 pravili (licitiranje pri bridžu je namreč precej zapleteno, ker služi tudi sporazumevanju med partnerjema). [13] Za igranje pa uporablja particijsko iskanje (ki je bilo razvito prav zanj in ga opisujem v razdelku 3.1.3) in si pomaga z vzorčenjem Monte Carlo (opisanim v razdelku 3.2.2), izboljšanim z metodo z dosegljivimi množicami (prav tako razvito zanj in opisano v razdelku 3.2.3). Že leta 1998 je zmagal na svetovnem prvenstvu v računalniškem bridžu in enako leta 2000. Po tistem je videti, da se ni udeležil nobenega vidnejšega tekmovanja. Najboljši ljudje so mu še vedno kos, čeprav jim je zelo blizu. Po tesnem porazu proti vrhunskima igralcema Michaelu Rosenbergu in Zii Mahmoodu leta 1998 je slednji preklical svojo ponudbo milijona funtov avtorju računalniškega programa, ki ga premega.

3.4.3. TAROK

Tarok je igra, ki je priljubljena le v nekaterih državah osrednje in vzhodne Evrope, ne pozna večjih organiziranih tekmovanj in o njem nisem zasledil nobenih teoretičnih študij. Zaradi tega tudi ni dosti programov za njegovo igranje. Obstajata pa dva slovenska izdelka: Tarok [21] in Tarok_MC [22]. Pravila igre so razložena v podpoglavju 4.1.

Program Tarok omogoča igranje treh ali štirih igralcov. Računalnikova igra se sicer zdi sprejemljiva, ima pa eno hudo pomanjkljivost: računalniški nasprotniki ne licitirajo, tako da če ne licitira človek, se vedno igra klop. To močno oteži preizkušanje, tako da je težko podati točno oceno. Program ima sicer svojo internetsko stran (ki je skrajno skopa), na e-pošto pa nihče ne odgovarja in glede na to, da je trenutna različica 1.0 in je iz leta 1995, se zdi, da se z njim nihče več ne ukvarja.

Tarok_MC je narejen za tri igralce. Računalnikova igra se zdi dokaj dobra in tudi licitiranje deluje. Občasno je sicer opaziti vprašljive poteze (denimo videti je, da računalnik vedno pobere, če more; včasih prepogumno meče kralje in pagata; opazil sem tudi, da je odvrgel pagata, ko mu še ni bilo treba in nikakor ni mogel vedeti, da ga ne bo pobral nasprotnik), a iz golega igranja je težko ugotoviti, kaj je razlog za napake (če to sploh so). Program ima spodobno internetsko stran in zdi se, da razvoj še ni zamrl, čeravno prav živahen tudi ni. Žal avtor ni dal pojasnil o njegovem delovanju in je izjavil, da nima časa za sodelovanje z menoj.

4. TAROK

4.I. PRAVILA TAROKA

Moj program je namenjen trem igralcem. Igra se tudi tarok za dva (ki pa je precej nepoznan) in za štiri (ki je dosti bolj zamotan). Za tarok za tri sem se odločil, ker je zanimivejši od taroka za dva in preprostejši od taroka za štiri in ter je med resnejšimi igralci najbolj priljubljen – Tarok zveza Slovenije [23] (po kateri sem povzel pravila) na svojih tekomvanjih uporablja to različico.

4.I.I. KARTE IN ŠTETJE

V igri se rabi 54 kart: 22 tarokov in po osem kart vsake barve. Taroki so označeni od ena do 21, zadnji pa je škis in šteje kot 22. Enica se imenuje pagat, 21 pa mond. V srcu in kari so karte od najnižje do najvišje štiri, tri, dve, ena, fant, kaval, dama in kralj, v piku in križu pa sedem, osem, devet, deset ter prav tako fant, kaval, dama in kralj. Najnižjim štirim kartam se reče plateljci.

Pri štetju se karte jemljejo po tri, vrednosti se seštejejo, nato pa se odšteje dve. Če na koncu ostane ena karta ali dve, se odšteje ena. Kot rezultat se šteje pobrana vrednost nad 35.

Karta	Vrednost
pagat, mond, škis in kralji	5
dame	4
kavali	3
fantje	2
ostalo	1

Tabela 1: Vrednosti kart

4.I.2. DELJENJE, LICITIRANJE, ZALAGANJE, NAPOVEDI

Vsak igralec dobi 16 kart, šest pa jih gre v talon (ki se ne pogleda do konca licitiranja). Prvi igralec nato napove, da bo igral tri. Naslednji lahko napove višjo igro (ali enako, če je na vrsti pred tistim, ki je licitiral zadnji) ali nič. Pri licitiranju zmaga igralec z najvišjo igro, potem ko se ostala dva odrečeta nadaljnjemu licitiranju. Če prvi ostane pri tri, lahko napove tudi klopa.

Igra	Vrednost
klop	
tri	10
dva	20
ena	30
solo	50
berač	70

Tabela 2: Vrednosti iger

Če igralec zmaga, se vrednost igre prišteje njegovemu rezultatu, sicer pa se odšteje. Pri klopu se vse pobrano šteje negativno, če pa kateri igralec pobere 35 točk ali več, se mu odšteje 70 točk. Pri beraču zmagovalec licitiranja ne sme pobrati ničesar, sicer izgubi.

Zmagovalec licitiranja se pri tri, dve ali ena založi. To pomeni, da se odkrije talon in se razdeli na po tri, dve ali eno karto. Eno izmed skupin vzame, ostanek pa pripade nasprotnikoma. Nato da iz svojega lista (to so karte, ki jih ima v roki) enako število kart med pobrane. Kraljev, pagata, monda ali škisa ni dovoljeno založiti, založene taroke (če jih kaj je) pa je treba pokazati soigralcem.

Potem zmagovalec licitiranja naznani, kaj meni, da bo lahko dosegel – to so napovedi.

Napoved	Vrednost
trula	20
kralji	20
pagat ultimo	50
valat	500

Tabela 3: Vrednosti napovedi

Trula pomeni, da bo pobral pagata, monda in škisa. Kralji pomenijo, da bo pobral vse kralje. Pagat ultimo pomeni, da bo zadnji vzetek pobral s pagatom. Valat pa pomeni, da bo pobral vse vzetke. Če napoved uresniči, se njena vrednost prišteje njegovemu rezultatu, sicer pa se odšteje. Če česa izmed naštetega ni napovedal, dosegel pa je vseeno, se njegovemu rezultatu prišteje polovična vrednost napovedi.

Na celotno igro ali na posamezne napovedi katerikoli nasprotnik lahko da kontro, kar pomeni, da se igra ali napoved šteje dvojno. Igralec na kontro lahko odgovori z re, nasprotnik na re s sub in igralec na sub z mort. To pomeni, da se dobljene ali igubljene točke početverijo, poosmerijo ali pošestnajsterijo. Ob neuspešni kontri nasprotnik, ki je ni napovedal, dobi toliko točk, kolikor je razlika med igro s kontro in igro brez kontre.

Pri klopu, beraču ali valatu vsi igralci dobijo radelce. Radelc igralcu podvoji rezultat (razen pri klopu) in se mu zbriše, kadar izgubi ali zmaga.

4.1.3. IGRANJE

Zmagovalec licitiranja začne igro (odvrže prvo karto), sledita pa mu druga dva igralca v izbrani smeri. Na barvo je treba vreči isto barvo in na tarok tarok; če igralec nima barve, mora odvreči tarok, če pa tudi tega nima, lahko vrže karkoli. Najvišja karta v barvi pobere vzetek, če ta ne vsebuje tarokov; če jih, pobere najvišji tarok. Če vzetek vsebuje pagata, monda in škisa, pobere pagat. Igralec, ki je vzetek pobral, je potem na vrsti. Pri klopu in beraču igralec mora vreči višjo karto, če jo ima. Pagata ne sme odvreči, če v to ni prisiljen. Pri klopu z vsakim pobranim vzetkom dobi še eno karto iz talona, dokler jih tam kaj je. Igra se konča, ko igralci odvržejo vse karte. Zmagovalec licitiranja zmaga, če pobere več kot 35 točk.

Od točk kateregakoli igralca, ki izgubi monda (ali pa ga pusti v talonu), se odšteje 21.

4.2. ALGORITMI

4.2.1. PREGLED

Ponujata se dva načina, kako se lotiti računalniškega programa za igranje iger: z rabo človeškega znanja in z grobo računsko silo. Pri prvi možnosti bi moral v program prenesti čim več človeškega znanja o taroku, na podlagi katerega bi ta potem igral. Pri drugi možnosti pa bi programu podal samo pravila igre, ta pa bi potem preiskoval poteze, ki jih lahko naredi, in nasprotnikove odgovore nanje ter se na podlagi te analize odločal, kako igrati.

Pretežna raba človeškega znanja ima kar nekaj slabosti. Prva je, da so potrebni ljudje, ki o izbranem področju veliko vedo – v mojem primeru denimo nisem imel nobenega stalno na voljo (sam sem sicer čisto spodoben tarokist, ravno mojster pa le nisem). Druga je, da tak program utegne biti predvidljiv. Nasprotnik lahko izrabi v svoj prid celo nekatere njegove dobre poteze, če jih pričakuje, slabe pa sploh – in gotovo bi se v program prikradla tudi kaka slaba. Tretja pa je neprilagodljivost – v nepričakovanem položaju program, ki se zanaša na vnaprej vdelano znanje, ne more vedeti, kako naj ravna. Poleg tega je nekatera znanja težko prenesti v program – zgledi so v razdelkih 4.2.2 in 4.2.3.

Glede na to, da sta pri igranju iger glavni prednosti računalnika pred človekom velika hitrost in popoln spomin, bi bilo nespametno, če ju ne bi izkoristil. Če bi bil računalnik zmožen v dovolj kratkem času preiskati vse drevo igre, bi to že zadoščalo. Vendar žal ni tako. Ker se temu cilju nisem zelo približal, se mi zdi dokaj verjetno, da težava ni samo v tem, da nisem izdelal dovolj dobrega preiskovalnega algoritma, ampak da je to za zdaj pretežak problem.

Tako sem bil primoran uporabiti kombinacijo obojega, čeprav s poudarkom na hitrem preiskovanju možnosti. Povsod, kjer sem se zatekel k znanju, sem skušal biti nekoliko konzervativen, tako da sem upošteval še kako možnost poleg tiste, za katero sem presodil, da je najboljša, in prepustil preiskovanju, da izbere. Poleg tega sem si prizadeval rabo znanja čim bolj omejiti (uporabljam ga na štirih mestih), ker ga je sicer težko dopolnjevati in iskati napake v njem. [18]

Področje umetne inteligence, mimo katerega ne morem, je strojno učenje. Uporabil sem ga le pri licitiranju in napovedovanju, pa še tam v zelo preprosti obliki. Razlog je, da nisem našel druge koristne rabe zanj, pa tudi zasledil nisem, da bi ga na tem področju uporabljal kdo drug.

Groba shema programa je takšna:

1. Licitiranje (opisano v razdelku 4.2.2): program preizkuša vedno težje igre, dokler ne pride do take, za katero oceni, da v njej ne more zmagati; oceno dobi tako, da za vsako igro tvori več možnih talonov ter se za vsakega založi in oceni dobljeni list.
2. Izbira kart iz talona, zalaganje in napovedi (opisano v razdelku 4.2.3): za vsako skupino kart, ki jo lahko vzame iz talona, se poizkusi čim bolje založiti, pri čemer s pomočjo človeškega znanja v grobem določi možnosti, nato pa jih temeljiteje preizkusi s simulacijo (odigra igro); njen rezultat mu služi kot ocena lista, pridobljenega z izbrano založitvijo, vsebuje pa tudi ugotovitve o možnostih za dodatne napovedi (trula, kralji, pagat ultimo).
3. Igranje (opisano v razdelku 4.2.4): uporabi vzorčenje Monte Carlo za tvorbo primerov nasprotnikovih listov in na njih uporabi različico algoritma alfa-beta, v katerem je človeško znanje uporabljeno v ocenjevalni funkciji, v transpozicijski tabeli in pri izločanju nekaterih manj zanimivih vej drevesa igre.

V programu sem moral določiti mnogo konstant. Nekatere pomenijo število vzorcev Monte Carlo, globino iskanja ipd. in so nastavljene tako, da naj ne bi nobeno opravilo, kot so zalaganje ali odločanje, katero karto vreči, trajalo več kot 10 sekund (na mojem računalniku, ki ima procesor Athlon XP P1800+ in 512 MB pomnilnika). To število je zgolj moja ocena, koliko potrpežljivosti se lahko pričakuje od igralca, in je včasih tudi nekoliko preseženo. Druge konstante so del človeškega znanja (konkretno mojega). Ker so vse zlahka prilagodljive, nekatere pa kdaj morda postanejo celo spremenljivke, ki jih bo lahko določal uporabnik, se mi je zdelo primerno posebej označiti jih: zapisal sem jih <v lomljenih oklepajih>.

4.2.2. LICITIRANJE

Licitiranje zahteva, da program oceni moč kart, ki jih ima v listu – da ugotovi, v kakšni igri se z njimi da zmagati. Seveda bi se to dalo storiti tako, da bi se na podlagi števila tarokov in kraljev, barv, katerih nima ali pa ima v njih malo kart (in je torej upravičeno domnevati, da bi se dale založiti), ter še česa izračunala ocena lista. Iz te ocene bi se nato določilo, kakšno igro je še moč dobiti, in program bi poizkusil izlicitirati takšno igro. Ljudje uporabljamo podoben postopek. Dejavniki, ki jih upoštevamo pri oceni lista, in pomen, ki jim ga pripisujemo, deloma izvirajo iz logičnega razmišljanja (ali vsaj nečesa, za kar si domišljamo, da je logično razmišljenje), deloma pa iz izkušenj. Npr. v prejšnjih igrah se je izkazalo, da osem tarokov, en kralj in odsotnost kart dveh barv zadoščajo za zmago. V trenutni igri imamo podobne karte, vendar imamo vse štiri barve, a v dveh le po eno karto. Pričakujemo, da bomo ti dve karti lahko založili in če igramo tri, iz talona ne bomo prisiljeni vzeti več kot eno karto, ki bi jo poleg njiju morali založiti, da bi dosegli list, ki nam je prej prinesel zmago. Program bi lahko ravnal enako: na podlagi izbranih dejavnikov bi sprva list ocenjeval po pravilih, ki bi jih določil človek, med igranjem pa bi s strojnim učenjem ta pravila izboljšal. A vendarle bi moral človek določiti, po katerih kriterijih naj list ocenjuje (oziroma bi bilo vsaj izredno težavno doseči, da bi tudi kriterije program našel sam), pa tudi učenje bi bilo bodisi dolgo (če bi bilo prilagodljivih mnogo parametrov) bodisi ne najbolj učinkovito (če bi bilo zastavljeno bolj togo). Zaradi opisanih težav sem, kot sem že omenil, stvar zastavil drugače.

Program po vrsti preizkuša vedno težje igre. Za vsako, kjer se zalaga, naključno tvori <5> talonov (če zalaganja ni, ta korak pač izpusti). Nato izbere karte iz talona, se založi in simulira igro (to je opisano v razdelku 4.2.3). Če je povprečen rezultat simuliranih iger dovolj velik, oceni, da preizkušano igro lahko dobi. Igre začne preizkušati pri tri. Če izgubi, preizkusi berača

in če izgubi tudi tega, ne licitira. Če tri dobi, preizkusi dve, ena in solo (dokler pač ne izgubi). Zadnjo igro, ki mu jo uspe dobiti, poizkusi izlicitirati.

Pri licitiranju gre v vsakem krogu kvečjemu tako visoko, da preseže druge igralce, in šele na koncu izbere najvišjo igro, za katero meni, da jo lahko dobi. V nasprotnem primeru namreč med licitacijo da nasprotniku več podatkov o svojem listu, kot bi bilo nujno potrebno, kar ne more biti koristno.

Ali je igro mogoče dobiti, bi se načeloma lahko ugotovilo po tem, ali je razlika med pobranimi kartami programa in nasprotnikov večja od nič. A praksa pokaže, da simulacija pomanjkljivo odseva dejansko igro, kar ni čudno, saj mora biti opravljena dosti hitreje in se dejanska igra lahko igra proti človeku, ki je očitno drugačen od simuliranega nasprotnika. Zaradi tega se za določanje minimalne razlike uporablja preprosta oblika strojnega učenja. Na začetku se minimalna razlika nastavi na <7>. Če nato program na licitaciji zmaga in igro izgubi, pomeni, da je bil preveč optimističen, zato se minimalna razlika poveča za ena. Če pa program na licitaciji ne zmaga in igro dobi, je to znak, da je bil preveč pesimističen, zato se razlika zmanjša za ena. Ostala dva primera sta znak, da je ravnal pravilno, zato se takrat razlika ne spremeni. Za berača se ravna podobno, le da se namesto minimalne razlike upošteva delež dobljenih iger. Začetna vrednost je <0,7>, po porazu se za 0,1 poveča, po <2> zaporednih zmagah pa za 0,1 zmanjša.

```
Licitiraj
  igra := klop
  dokler (razlika >= minimalna vrednost) & (igra < solo)
    igra := igra + 1
    razlika := 0
    ponovi <5>-krat
      naključno razdeli karte
      razlika := razlika + Izberi-založi-simuliraj (karte, igra)
    razlika := razlika / <5>
  če razlika < minimalna vrednost
    igra := igra - 1;
  če igra = klop
    zmage := 0
    ponovi <5>-krat
      naključno razdeli karte
      če Izberi-založi-simuliraj (karte, berač) = -70
        zmage := zmage + 1
    če zmage / <5> >= minimalni delež zmag
      igra := berač
  vrni igra
```

Algoritem 4: Licitiranje

Algoritem 4 kaže funkcijo, ki določi najvišjo igro, ki jo je moč dobiti.

4.2.3. IZBIRA KART IZ TALONA, ZALAGANJE, NAPOVEDI

Kadar je ena skupina kart v talonu bistveno boljša od druge, se ni težko odločiti, katero vzeti. Npr. trije taroki so prav gotovo boljši od treh plateljcev – tako za človeka kot za program je tu izbira lahka. V splošnem pa je treba upoštevati, kakšen bo list, ko mu bomo dodali izbrane karte in se potem založili. Npr. srčeva dama in tarok se zdita boljša od dveh pikovih ali dveh križevih plateljcev. A če imamo v srcu samo kavala, ki ga nameravamo založiti skupaj z edino

karo, v piku pa imamo že tri karte, je verjetno bolje vzeti dva pikova plateljca. Če vzamemo pikova plateljca, lahko založimo tako srce kot karo, pikove karte pa bi nam v vsakem primeru ostale. Da bi program tako sklepal eksplicitno, ni enostavno doseči, zato sem se, kot sem že omenil, problema lotil drugače.

Program preizkusi vse skupine kart iz talona, se vsakič založi in nato simulira igro. Tista skupina, pri kateri je rezultat igre najboljši, je izbrana.

Preden opišem, kako poteka zalaganje, naj pojasnim dva izraza. Kombinacija barv so karte, ki se dajo založiti, tako da v listu katere barve nimamo ali pa imamo v njej samo kralja. Kombinacija barv lahko šteje tudi manj kart, kot jih moramo založiti. Končna kombinacija pa je toliko kart, kolikor jih res moramo založiti.

Pri zalaganju program najprej poišče kombinacije barv. Vseh končnih kombinacij je namreč preveč, da bi jih lahko pregledal (če naj vsako temeljito ovrednoti). Poleg tega je malo dvoma, da je dobro, če katere izmed barv nimamo ali pa imamo v njej samo kralja. V obeh primerih imamo možnost pobrati vse karte te barve, ocenil pa sem, da je vendarle bolje imeti kralja. V nasprotnem primeru se namreč pogosto pripeti, da tisti, ki kralja ima, najde priložnost, da ga pobere soigralec, poleg tega pa nasprotnika neredko predpostavita, da če imamo kralja, imamo še kako karto v isti barvi (in sta zato neprevidna z npr. damo). Velja tudi upoštevati, koliko so vredne karte, ki jih bomo založili. Takole se izračuna začetna ocena kombinacije barv:

$$\mathrm{f\,(kombinacija\_barv)} = \sum_{i=1}^{n} <10> + \mathrm{kralj}_i \times <3> + \mathrm{založeno}_i \times <1> \qquad (3)$$

V enačbi (3) je n število barv, ki so v kombinaciji_barv založene, kralj_i označuje prisotnost kralja v i-ti barvi, $\mathrm{založeno}_i$ pa je vrednost založenih kart i-te barve.

Ker tu očitno uporabljam človeško znanje, ne izberem le najboljše kombinacije, ampak program opravi pri najboljših <2> kombinacijah barv nadaljnjo analizo. Slednja tudi pove, kako naj kombinacijo barv dopolni do končne kombinacije, če je to potrebno. Pri dopolnilnih kartah pa se preizkusijo vse (veljavne) možnosti. Za vsako se potem ostale karte naključno razdelijo med nasprotnika in opravi se simulacija. Če je končnih kombinacij veliko, ni časa, da bi se za vsako opravilo več simulacij z različnimi razdelitvami kart, pa tudi potrebno ni, ker se med seboj ponavadi ne razlikujejo bistveno. Če jih ni veliko, pa se pri dokončnem zalaganju vseeno opravi simulacija z vsaj <20> različnimi razdelitvami kart, pri poizkusnem med licitiranjem pa s <5> (to zveni malo, a upoštevati je treba, da se jih toliko opravi za vsakega izmed <5> naključno tvorjenih talonov). Končna ocena kombinacije barv se izračuna takole:

$$\mathrm{f\,(kombinacija\_barv)} = \frac{\sum_{i=1}^{m} \mathrm{vk}_i + \mathrm{vnk} \times <2>}{\mathrm{m} + <2>} \qquad (4)$$

V enačbi (4) je m število končnih kombinacij, ki nastanejo iz kombinacije_barv, vk_i je vrednost i-te končne kombinacije, vnk pa je vrednost najboljše končne kombinacije.

Določi se najboljša kombinacija barv, ki se razširi v najboljšo končno kombinacijo, njena vrednost pa se pripiše skupini kart iz talona.

Izbiro najboljše založitve za dani list, razširjen z eno izmed skupin kart iz talona, kaže algoritem 5.

Pri preizkušanju kombinacij barv se tudi šteje, kolikokrat je program dobil trulo, kralje ali pagat ultimo. Če se je kaj od tega zgodilo v zadostnem deležu primerov, to tudi napove. Načeloma bi zadoščal uspeh v polovici primerov, a ker rezultatom simulacije ne gre povsem

```
Založi-simuliraj (list + del talona)
   tvori kombinacije barv
   sortiraj kombinacije barv po začetni oceni
   (naj list, vrednost_nl) := (nil, -∞)
   za min (število kombinacij barv, <2>) kombinacij barv
      (naj list kombinacije, vrednost_nlk) := (nil, -∞)
      Išči-končno-kombinacijo (trenutna kombinacija barv)
      če vrednost_nlk > vrednost_nl
         (naj list, vrednost_nl) := (naj list kombinacije, vrednost_nlk)
   vrni (naj list, naj vrednost)

Išči-končno-kombinacijo (kombinacija)
   če kombinacija ne vsebuje dovolj kart
      za vse veljavne karte
         kombinacija := kombinacija + trenutna veljavna barva
         Išči-končno-kombinacijo (kombinacija)
   sicer
      list kombinacije := list + del talona - kombinacija
      vrednost_lk := Simuliraj (list kombinacije)
      če vrednost_lk > vrednost_nlk
         (naj list kombinacije, vrednost_nlk) := (list kombinacije,
           vrednost_lk)
```

Algoritem 5: Zalaganje

zaupati, se ti deleži prilagajajo na podoben način kot pri izbiri igre. Začetni deleži so <0,5> za trulo (ker je odvisna predvsem od razdelitve kart in manj od igranja), <0,6> za kralje in <0,7> za pagat ultimo (ker je najzahtevnejši in najbolj odvisen od dobrega igranja). Če program kaj od tega napove in se napoved ne uresniči, se ustrezni delež poveča za 0,1, po <2> zaporednih uspešnih napovedih pa se zmanjša za 0,1.

4.2.4. IGRANJE

Jedro programa je algoritem alfa-beta. Posebnost njegove uporabe pri taroku je, da je treba izrecno ugotavljati, kakšne vrste je naslednje vozlišče – v mnogih igrah se namreč *min* in *maks* izmenjujeta, pri taroku pa ni tako. Če je program zmagal na licitiranju, je 1/3 vozlišč vrste *maks* (ko je na potezi program), sicer pa 2/3 (ko ni na potezi igralec, ki je zmagal na licitiranju), pa tudi vedno v enakem vrstnem redu ne nastopajo. Ker mi ni uspelo implementirati particijskega iskanja, sem poizkusil z vsemi ostalimi pomembnejšimi izboljšavami.

Najprej sam dodal **transpozicijsko tabelo** (opisano v razdelku 3.1.2). Implementiral sem jo kot zgoščeno tabelo z <1.000.003> elementi. Kadar je treba element vpisati v zasedeno polje, starega kratko malo prepišem, kot se pri transpozicijskih tabelah navadno ravna (sploh pa se to ne zgodi pogosto, ker je tabela precej velika). Za zgoščevalno funkcijo sem uporabil:

$$\text{indeks}(\text{stanje}) = \left(\sum_{i=0}^{2} \text{stanje}_i \times <101>^i\right) \bmod <1000003> \qquad (5)$$

V enačbi (5) je stanje_i opis kart i-tega igralca. <1.000.003> in <101> sta praštevili, katerih raba pri zgoščevalnih funkcijah je nasploh priporočljiva [24], pa tudi v praksi sta se izkazali veliko bolje od 'lepših' števil – npr. pri vrednostih 100 in 1.000.000 se preišče približno 2,6-krat več vozlišč, ker se mnogo polj v tabeli prepiše.

Pri kodiranju listov posameznih igralcev sem uporabil osnovno idejo particijskega iskanja (opisanega v razdelku 3.1.3): razlik med listi, ki so se mi zdele zanemarljive, nisem upošteval. Tu je še eno mesto, kjer sem uporabil človeško znanje – takole:

$$\begin{aligned} \text{stanje}_i &= \text{k\_barv}_i + \text{k\_tarokov}_i \times \text{nk\_barv} + \text{k\_vzetka}_i \times \text{nk\_lista} \\ \text{k\_barv}_i &= \sum_{j=0}^{3} (\text{k\_plateljcev}_{ij} + \text{k\_figur}_{ij} \times \text{nk\_plateljcev}) \times \text{nk\_barve}^j \\ \text{k\_tarokov}_i &= \text{št\_tarokov}_i + \frac{\text{vsota\_tarokov}_i}{<10>} \times 16 + \text{visoki\_taroki}_i \times \text{nk\_vsote\_tarokov} \times 16 \end{aligned} \tag{6}$$

V enačbah (6) je za i-tega igralca k_plateljcev_{ij} število plateljcev, ki jih ima v j-ti barvi, k_figur_{ij} natančno opiše njegove figure j-te barve, št_tarokov_i in vsota_tarokov_i sta število in vsota tarokov, ki jih ima, visoki_taroki_i natančno opiše, kateri je njegov najvišji tarok ter ali ima pagata, monda in škisa, k_vzetka_i pa pove, katero karto je nazadnje odvrgel (če še ni bila pobrana); nk_nečesa pomeni največjo kodo nečesa – te vrednosti so zbrane v tabeli 4.

Oznaka	**Vrednost**
nk_barv	40.960.000
nk_lista	242.210.560.000
nk_plateljcev	4
nk_barve	80
nk_vsote_tarokov	21

Tabela 4: Največje kode

Kot kažejo enačbe (6), so podatki, ki sem jih zanemaril, katere plateljce igralec ima (upoštevam samo število) in natančno katere taroke ima (upoštevam samo število, vsoto na <10> natančno, najvišji tarok ter pagata, monda in škisa). To zmanjša število preiskanih vozlišč za približno 2,3-krat.

Vdelal sem tudi **zgodovinsko hevristko** (opisano v razdelku 3.1.4). Poteze, za katere vodim zgodovino uspešnosti, so karte za vsakega igralca. Na začetku iskanja vsem potezam nastavim zgodovinske vrednosti na nič. Kadar poteza povzroči rez, ji vrednost povečam takole:

$$\text{nova\_vrednost} = \text{stara\_vrednost} + 2^{\text{globina}} \tag{7}$$

Tako enačbo (7) kot tudi odločitev, da se vrednost poveča le ob rezu, sem povzel po [10]. Preizkusil pa sem tudi možnost, da zgodovinsko vrednost poteze povečam vsakič, ko se v nekem vozlišču izkaže za najboljšo – bodisi za enako kot takrat, ko povzroči rez, bodisi za manj. Obe možnosti sta se izkazali za nekoliko slabši.

Uporabljam tudi **iskanje z najmanjšim oknom** (opisano v razdelku 3.1.5). Glede na to, da z zgodovinsko hevristiko razvrščam poteze, je bilo pričakovati, da bo dobro delovalo. Zanimivo pa je, da dobro deluje tudi, če razvrščanje potez izključim, kar se vidi v razdelku 4.4.1.

In navsezadnje sem se odločil **izločati nekatere poteze**, kar je oblika prilagajanja globine iskanja (o katerem pišem v razdelku 3.1.6). Prav gotovo so v vsakem položaju poteze, ki bi jih

s človeškim znanjem lahko izločil kot nedvomno slabe, a povsod je to precej težavno storiti. Zato sem se omejil na tretje karte vzetka – ko sta ostala dva igralca že odvrgla svoji karti, je na voljo dovolj informacij za umno odločitev.

Karte, za katere sem določil, da jih utegne biti vredno obdržati, so tele (v oklepajih so okoliščine, v katerih so uporabne pri navadni igri in pri klopu; pri klopu je treba upoštevati, da se izbira samo med dovoljenimi kartami – se pravi takimi, ki vzetek poberejo, če jih igralec ima):

1. najnižja (navadna igra: kadar igralec ne bo pobral; klop: kadar bi rad pobral s čim manj vredno karto – v poštev pride pri barvah);
2. najnižja, ki še pobere (navadna: kadar želi pobrati in za to uporabiti čim nižjo karto – v poštev pride pri tarokih; klop: nikoli);
3. najvišja (navadna igra: kadar želi, da on ali njegov soigralec pobere čim več vredno karto – v poštev pride pri barvah; klop: če mora tako ali tako pobrati, naj to stori z največ vredno karto, da ne bo z njo prisiljen pobrati kdaj v prihodnje);
4. druga najvišja (navadna igra: kadar želi, da on ali njegov soigralec pobere čim več vredno karto, a želi najvišjo prihraniti, da bo z njo pobral kasneje – v poštev pride pri barvah in le če je najvišja vsaj <dama>; klop: če mora tako ali tako pobrati, a ne želi z najvišjo, ker je ta mond in bi rad, da ga nekdo pobere s škisom);
5. drugi najnižji tarok za pagatom (navadna igra: kadar ne bo pobral in je najnižja karta pagat; klop: nikoli);
6. mond (navadna igra: treba ga je obravnavati posebej, ker je veliko vreden, ni pa nujno, da bo z njim pobral; klop: nikoli).

Kadar igralec nima prve barve vzetka ali taroka, je treba obravnavati vsako barvo posebej, sicer pa le eno. Pri beraču vedno pride v poštev le najvišja karta.

Upoštevam pet vrst stanj igre (vsako od njih pa ima dve različici – kjer trenutno kaže, da bo pobral nasprotnik, in kjer je soigralčeva karta višja od nasprotnikove): igralec vrže barvo na isto barvo, tarok na barvo, drugo barvo na barvo (ker prave nima), tarok na tarok in barvo na tarok (ker taroka nima). Tabela 5 kaže, katere karte se ne izločijo v katerih stanjih pri navadni igri, tabela 6 pa kaže isto za klopa in berača.

Stanje / Karta		Najnižja	Še pobere	Druga najvišja	Najvišja	Najnižja za pagatom	Mond	Po barvah
Soigralčevo	barva na barvo			✓	✓			
	tarok na barvo	✓	✓				✓	
	druga barva na barvo			✓	✓			✓
	tarok na tarok	✓	✓				✓	
	barva na tarok			✓	✓			✓
Nasprotnikovo	barva na barvo	✓		✓	✓			
	tarok na barvo	✓	✓			✓	✓	
	druga barva na barvo	✓						✓
	tarok na tarok	✓	✓			✓	✓	
	barva na tarok	✓						✓

Tabela 5: Karte, ki se ne izločijo pri navadni igri

Stanje / Karta		Najnižja	Še pobere	Druga najvišja	Najvišja	Najnižja za pagatom	Mond	Po barvah
Klop	barva na barvo	✓			✓			
	tarok na barvo	✓		✓	✓			
	druga barva na barvo				✓			✓
	tarok na tarok	✓		✓	✓			
	barva na tarok				✓			✓
Berač	barva na barvo	✓			✓			
	tarok na barvo	✓			✓			
	druga barva na barvo				✓			✓
	tarok na tarok	✓			✓			
	barva na tarok				✓			✓

Tabela 6: Karte, ki se ne izločijo pri klopu in beraču

Ocenjevalna funkcija je še zadnji del, ki temelji na človeškem znanju. Če bi drevo igre lahko preiskal do konca, bi bila najboljša ocena kar rezultat igre, ker pa ga ne morem, je vanjo treba vdelati tudi taktično vrednotenje stanja igre. Poleg tega je treba paziti, da ocenjevalna funkcija ni preveč kompleksna, ker se kliče precej pogosto in bi v tem primeru lahko resno upočasnila program. Takole se računa pri navadni igri (ne pri klopu ali beraču):

$$\begin{aligned} f(\text{stanje}) = {} & \text{razlika} \times \langle 5\rangle + \text{vsota\_tarokov} \times \langle 1\rangle + \text{mond} \times \langle 5\rangle \times \langle 10\rangle + \text{škis} \times \\ & \times \langle 5\rangle \times \langle 10\rangle + \text{pagat\_ultimo} \times \text{napoved\_pagata\_ultimo} \times \langle 5\rangle \times \langle 25\rangle \end{aligned} \tag{8}$$

V enačbi (8) razlika označuje razliko med vrednostjo pobranih kart programa in njegovega morebitnega soigralca ter nasprotnikov – razlog za ta element je očiten. Z vsota_tarokov je označena vsota programovih tarokov in preprečuje preveč pogumno odmetavanje tarokov. Brez tega se je namreč dogajalo, da je imel visoke taroke, katerih uporaba je bila dobro ocenjena, ker nasprotnik ni imel višjih, zato jih je hitro porabil, potem pa je nasprotnik s svojimi nižjimi, ki so mu ostali, pobral veliko vredne karte. Spremenljivki mond in škis označujeta prisotnost teh dveh kart v programovem listu. Ker sta karti sami veliko vredni in ker se z njima navadno pobere, se je brez tega redno dogajalo, da ju je program uporabil ob prvi priložnosti, namesto da bi ju bil prihranil za kako pomembnejšo. V stanjih, kjer je program zadnji vzetek pobral s pagatom, je pagat_ultimo 1, v stanjih, kjer mu ga je v zadnjem vzetku pobral nasprotnik, je –1, drugje pa je 0; Če je pagat ultimo napovedal, je napoved_pagata_ultimo 2, sicer pa 1.

Pri klopu si mora program seveda prizadevati, da bi čim manj pobral. Obenem pa je dobro, če nasprotniki poberejo čim več, kajti več veliko vrednih kart ko je izločenih iz igre, manj nevarnosti je, da bo program prisiljen katero pobrati. In seveda je zaželeno imeti čim nižji najvišji tarok, da bodo nasprotniki imeli višje. Se pa temu ne sme dati prevelikega poudarka, ker sta prva dva kriterija pomembnejša. Ocenjevalna funkcija je zato takšna:

$$\begin{aligned} f(\text{stanje}) = {} & -\text{pobrano\_programa} \times \langle 100\rangle + \text{pobrano\_nasprotnikov} \times \langle 20\rangle + \\ & \text{najvišji\_tarok} \times \langle 1\rangle \end{aligned} \tag{9}$$

Pri beraču je ocenjevalna funkcija zelo preprosta. Če ga igra program, je njena vrednost $-\infty$ v stanjih, kjer je kaj pobral, sicer pa 0. Če pa ga igra njegov nasprotnik, je njena vrednost ∞ v stanjih, kjer je nasprotnik kaj pobral, sicer pa 0.

Preiskovanje drevesa igre je prikazano v algoritmu 6 – v primerjavi z algoritmom 2 (alfa-beta s transpozicijsko tabelo) so rdeče označeni deli, povezani z zgodovinsko hevristiko, vijoličasto deli, povezani z iskanjem z minimalnim oknom, in modro deli, povezani z izločanjem nekaterih vej drevesa igre.

Pri običajni igri program uporablja **vzorec Monte Carlo** z <20> razporeditvami kart. Na začetku išče do globine <9>; ko imajo igralci še po <10> kart, se drevo igre dovolj zoži, da se to lahko poveča na <12>, pri <7> kartah pa se poveča na <15>. Te globine pa veljajo le za igralca, ki vrže prvo karto iz vzetka; za drugega in tretjega so za ena ali dve manjše, ker je smiselno preiskovati le do konca vzetka. Pri tvorbi vzorčnih razporeditev kart se upoštevajo vsi gotovi podatki, ki jih ima program o kartah drugih igralcev: da ne moreta imeti kart, ki jih ima program sam, ki jih je kdo odvrgel, ki so v talonu (če je njegova vsebina znana) in ki jih je kdo dobil iz talona (pri klopu); da nimata barve, na katero sta vrgla tarok; ter da nimata tarokov, če sta na barvo vrgla drugo barvo ali na tarok nista vrgla taroka.

```
Alfa-beta-tarok (vozlišče, alfa, beta)
  če je vozlišče list
    vrni njegovo vrednost
  sicer
    a := alfa
    b := beta
    najboljša karta := -1
    če je vozlišče v transpozicijski tabeli
      če je vrste
        točno: vrni vrednost iz tabele
        spodnja meja: a := vrednost iz tabele
        zgornja meja: b := vrednost iz tabele
    izloči nezanimive naslednike vozlišča
    razvrsti naslednike vozlišča glede na zgodovinsko hevristiko
    če je vozlišče vrste min
      za vse naslednike vozlišča
        vrednost := Alfa-beta-tarok (trenutni naslednik vozlišča, a, b)
        če (vrednost <= a) & (vrednost > alfa)
          vrednost := Alfa-beta-tarok (trenutni naslednik vozlišča,
            alfa, b)
        b := min (b, vrednost)
        a := max (a, b - 1)
        če b <= a
          najboljša karta := trenutni naslednik vozlišča
          prekini zanko
      rezultat :=  b
    sicer
      za vse naslednike vozlišča
        vrednost := Alfa-beta-tarok (trenutni naslednik vozlišča, a, b)
        če (vrednost >= b) & (vrednost < beta)
          vrednost := Alfa-beta-tarok (trenutni naslednik vozlišča, a,
            beta)
        a := max (a, vrednost)
        b := min (b, a + 1)
        če a >= b
          najboljša karta := trenutni naslednik vozlišča
          prekini zanko
      rezultat := a
    vrsta := točno
    če vrednost >= b
      vrsta := spodnja meja
    če vrednost <= a
      vrsta := zgornja meja
    shrani v transpozicijsko tabelo (vozlišče, rezultat, vrsta)
    če najboljša karta >= 0
      popravi zgodovinsko tabelo (najboljša karta, trenutni igralec,
        trenutna globina)
    vrni rezultat
```

Algoritem 6: Iskalni algoritem za tarok

Za vsako razporeditev kart se poleg iskanja do največje globine išče še do globine enega vzetka. Rezultati tega iskanja pri končni odločitvi veljajo <0,7>-krat toliko kot rezultati iskanja do polne globine. Zakaj je to potrebno, pokaže primer: vsak igralec ima še dve karti. Program ima škisa in srčevega kralja, ostala igralca pa imata dva taroka in dve drugi nesrčevi karti, a program ne ve, kdo ima kaj. Vzorčenje Monte Carlo predpostavi, da ima eden oba taroka in drugi ostali karti (to je le poenostavljan primer – v resnici je zelo malo verjetno, da ne bi v nobenem vzorcu imel vsak nasprotnik po enega taroka). Program je na potezi. Zdi se, da je

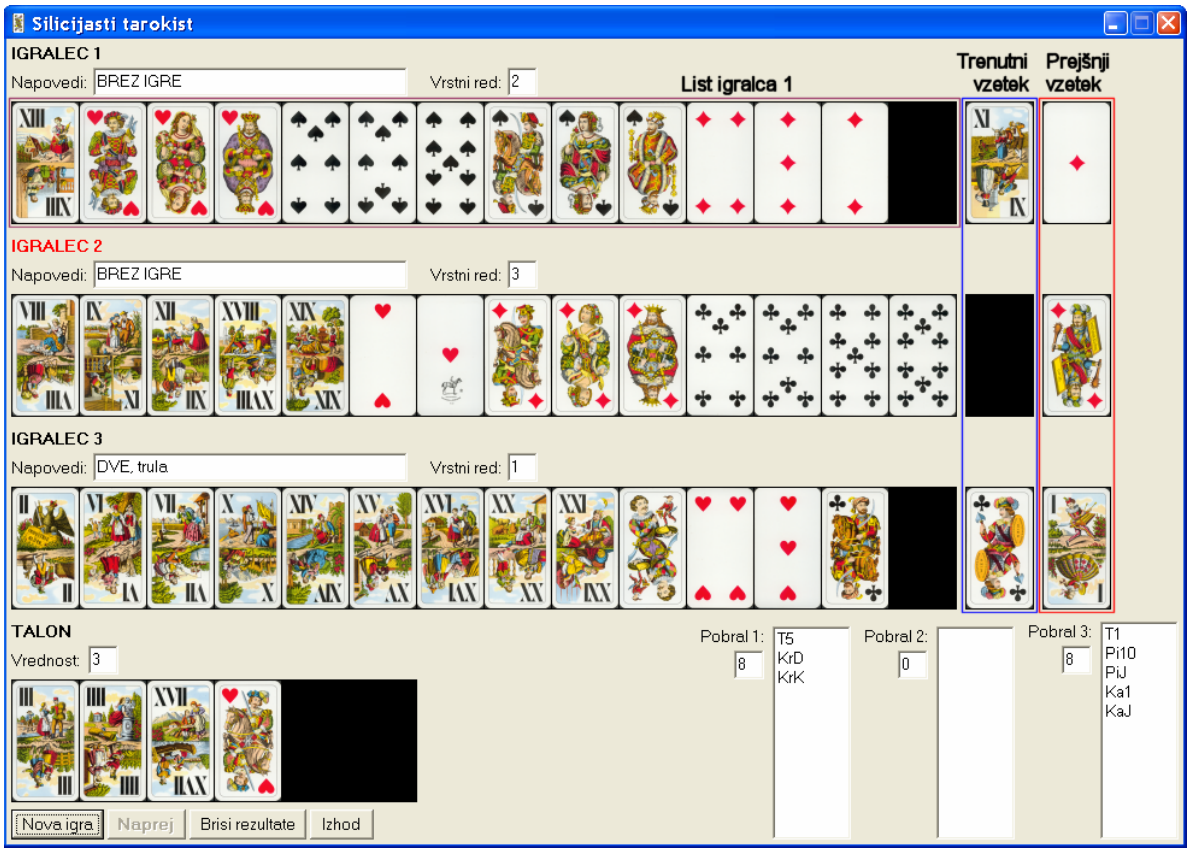
vseeno, ali najprej odvrže škisa in potem kralja – kralj je v vsakem primeru izgubljen. Brez plitvejšega iskanja bi se torej lahko zgodilo, da bi najprej odvrgel kralja. Če bi potem oba nasprotnika odvrgla taroka, bi postalo očitno, da je bila ta odločitev slaba. Plitvejše iskanje poudari takojšnjo koristnost potez in bi v tem primeru preprečilo napako. Pri preiskovanju do globine enega vzetka je namreč bistveno bolj dobičkonosno najprej odvreči škisa.

Pri simulaciji, kakršna se uporablja med licitiranjem ter izbiro kart iz talona in zalaganjem, pa je globina iskanja le tri (oziroma dve in ena), ker bi sicer trajala predolgo.

4.3. UPORABA PROGRAMA

4.3.1. GLAVNO OKNO

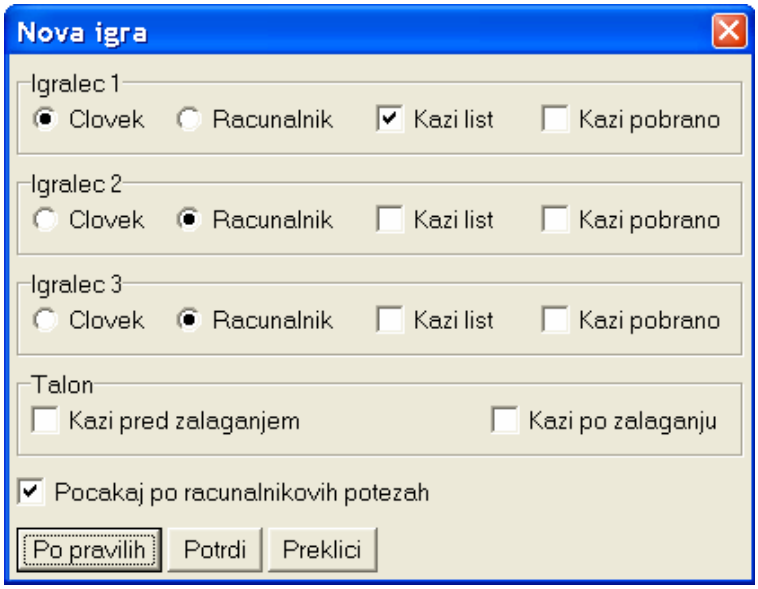
Kot kaže slika 9, je v glavnem oknu za vsakega igralca prikazano, kakšno igro igra in katere dodatne napovedi je naznanil ('Napovedi') ter kateri po vrsti je pri trenutnem vzetku ('Vrstni red'). Lahko je prikazan tudi njegov list, vsekakor pa je desno od lista karta, ki jo je zadnjo odvrgel, desno od nje pa njegova karta prejšnjega vzetka. Če se igra berač, se vidi ne le zadnja karta, ampak vse odvržene. Spodaj se lahko vidita talon in njegova vrednost ('Vrednost'). Prikazane so lahko tudi pobrane karte igralcev ('Pobral 1–3') in njihove vrednosti (v okencih pod napisi). Tu so karte označene ne s sličicami, ampak z napisi: barve so Sr (srce), Ka (karo), Pi (pik) in Kr (križ), tarok pa T; sledi jim številka ali oznaka figure: J (fant), C (kaval), D (dama) in K (kralj). Trenutni igralec je označen rdeče.



Slika 9: Glavno okno

Človeški igralec izigra karto tako, da klikne na njeno sličico. S pritiskom na 'Nova igra' se začne nova igra (več o tem v naslednjem razdelku). Če je nastavljeno, da se po računalnikovi potezi počaka, se igra nadaljuje s pritiskom na 'Naprej'. 'Briši rezultate' nastavi rezultat vseh odigranih iger na nič in za delilca določi igralca 1. S pritiskom na 'Izhod' pa se zapusti program.

4.3.2. NOVA IGRA

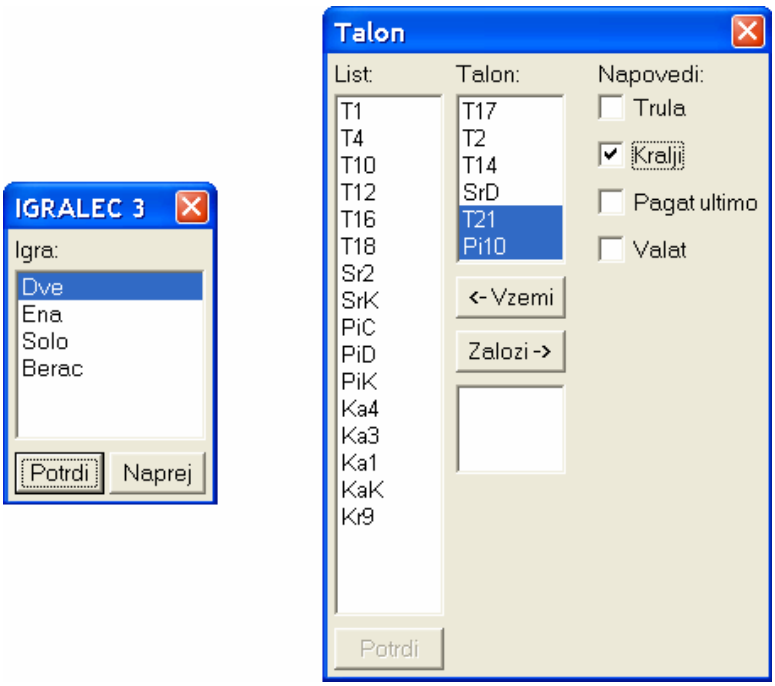


Slika 10: Nova igra

Kot kaže slika 10, se za vsakega igralca da določiti, ali bo človek ali računalnik. S 'Kazi list' in 'Kazi pobrano' se nastavi, ali bodo njegov list in pobrane karte vidni. Za talon se s 'Kazi pred zalaganjem' in 'Kazi po zalaganju' nastavi, ali je viden takoj po razdelitvi kart in ali je viden med igro (med zalaganjem je viden vedno, če se igrá igra, pri kateri se zalaga). 'Pocakaj po racunalnikovih potezah' pomeni, da računalnik po vsaki potezi (napovedi, založitvi, odvrženi karti ...) počaka, da igralec pritisne na 'Naprej'. Če igrajo trije računalniki, se s 'Po pravilih' vse nastavi na vidno, če pa igra kak človek, se listi vseh ljudi nastavijo na vidne, vse ostalo pa se skrije. 'Potrdi' začne novo igro in 'Preklici' zapre okno, ne da bi se igra začela.

4.3.3. LICITIRANJE, ZALAGANJE, NAPOVEDI

Ko se začne nova igra, človeški igralci licitirajo, kot je prikazano na sliki 11 levo: izbirajo lahko med vsemi igrami, ki so trenutno na voljo, z 'Naprej' pa se odrečejo licitaciji. Gumb 'Naprej' je onemogočen pri prvem igralcu, ker mora igrati vsaj tri, in na koncu, ko se mora zmagovalec licitacije odločiti, kaj bo res igral.



Slika 11: Licitiranje ter izbira kart iz talona, zalaganje in napovedi

Sledijo izbira kart iz talona in zalaganje ter napovedi. Ustrezno okno kaže slika 11 na desni. Stolpec 'List' so karte, ki jih ima igralec v roki, stolpec 'Talon' pa je talon. V obeh je moč izbirati karte (v talonu le v skupinah, ki so v skladu z igrano igro) in jih z dvoklikom ali pritiskom na 'Vzemi' oziroma 'Založi' prenesti v list oziroma jih založiti. Desno se lahko izberejo napovedi. Po pritisku na 'Potrdi' se igranje začne.

4.3.4. DRUGO

Slika 12: Druga okna Silicijastega tarokista

Če kak igralec založi taroke, so ostali o tem obveščeni, kot kaže slika 12 levo. Obveščeni so tudi o kartah, ki jih pri klopu pri prvih šestih vzetkih igralci dobijo iz talona – to se vidi na sliki 12 na sredini. Ob koncu igre pa se odpre okno z rezultati zadnje igre, skupnimi rezultati vseh iger, ki so bile odigrane od takrat, ko je bil bodisi program pognan bodisi je bil pritisnjen gumb 'Brisi rezultate', in številom radelcev, ki jih ima vsak igralec. Vodeči je označen rdeče. To kaže slika 12 desno.

4.4. OCENA PROGRAMA

4.4.1. MERITVE

Da se pokaže, kolikšni so prispevki posamičnih izboljšav algoritma alfa-beta in njihovih kombinacij, sem preštel, koliko vozlišč se razvije pri enem iskanju, in izmeril, koliko časa je potrebnega zanj. Nato sem še izračunal, koliko časa se porabi na vozlišče, da se vidi, koliko pribitka pri vsakem vozlišču povzroči katera izboljšava. Iskanje sem pognal z vsemi izboljšavami, z vsemi zboljšavami razen ene (vse štiri možnosti), s samo po eno izboljšavo (vse štiri možnosti) in z golim algoritmom alfa-beta. Rezultati veljajo za prvo karto, igrano v igri, in so povprečje 20 iger z vzorcem Monte Carlo velikosti 20 (kot se uporablja tudi pri običajnem igranju) pri globini iskanja devet (kot se na začetku uporablja tudi pri običajnem igranju). Časi so bili izmerjeni na računalniku s procesorjem Athlon XP P1800+ in 512 MB pomnilnika.

Algoritem	Vozlišča	Čas (s)	Čas/vozlišče (µs)
vse izboljšave	8.667	0,248	28,6
brez transpozicijske tabele	25.349	0,527	20,8
brez zgodovinske hevristike	13.467	0,350	26,0
brez najmanjšega okna	15.827	0,433	27,4
brez izločanja nekaterih vej	15.952	0,435	27,3
transpozicijska tabela	95.575	2,451	25,6
zgodovinska hevristika	324.633	5,463	16,8
najmanjše okno	320.792	4,438	13,8
izločanje nekaterih vej	258.060	4,891	18,9
nič izboljšav	1.598.924	21,266	13,3

Tabela 7: Učinkovitost izboljšav algoritma alfa-beta

Kot se lahko razbere iz tabele 7, so vse izboljšave zelo koristne. Vse štiri skupaj povzročijo, da se preišče 184-krat manj vozlišč, kot bi se jih z golim alfa-beta, za kar se porabi 86-krat manj časa (kar je posledica tega, da vse izboljšave podaljšajo čas, ki se porabi za eno vozlišče, za malo več kot dvakrat). Očitno je, da ima največji učinek transpozicijska tabela, celo če upoštevamo, da najbolj podaljša čas za eno vozlišče. Ostale tri izboljšave so si precej podobne, čeravno zgodovinska hevristika je nekoliko slabša od ostalih dveh.

Zanimivi so tudi časi/vozlišče. Kot sem že omenil, jih transpozicijska tabela najbolj poveča. Za ostale izboljšave iskanja brez po ene izboljšave kažejo, da so si precej podobne in povzročajo zanemarljive pribitke. Pri iskanjih s po eno izboljšavo pa je slika drugačna: tu najmanjše okno čas/vozlišče še vedno podaljša zelo malo, ostali dve izboljšavi pa ga kar opazno. Prav dobro tega ne znam pojasniti, a možna razlaga bi bila, da je pri najmanjšem oknu in pri golem

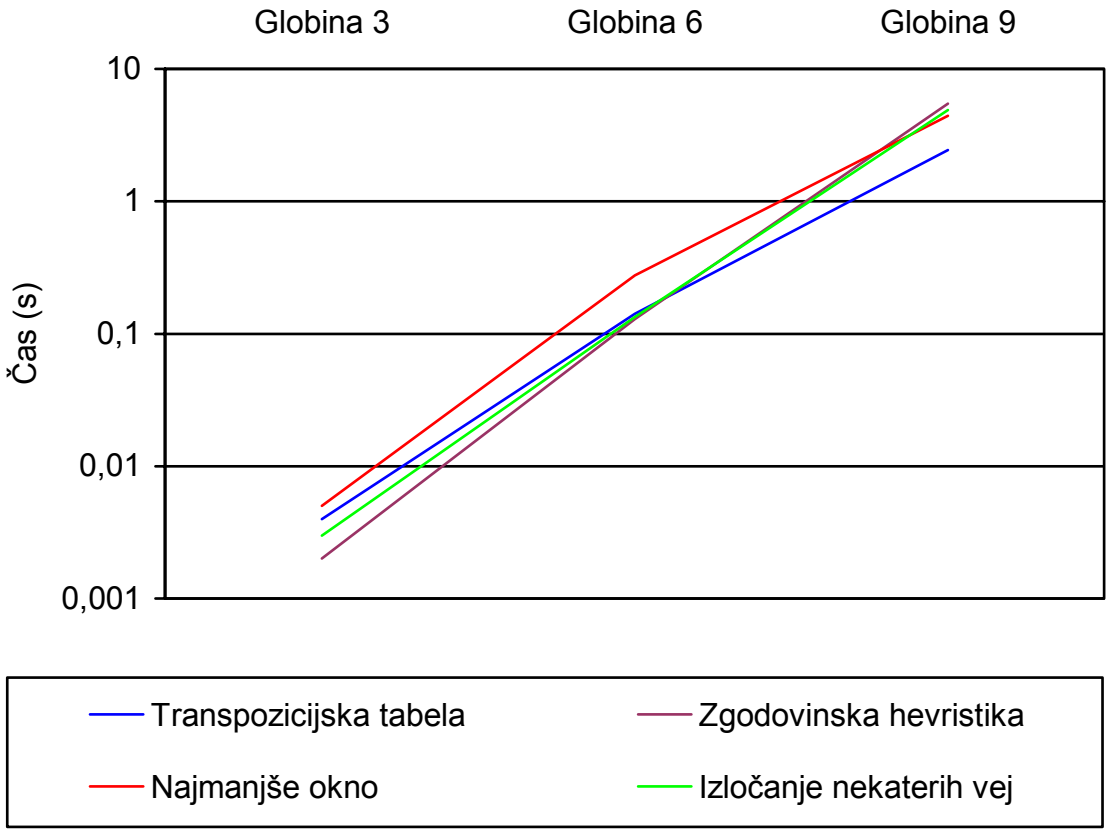
alfa-beta drevo ožje blizu korena in širše na dnu, tako da je veliko vozlišč končnih (taka se obdelajo hitreje).

Želel sem tudi ugotoviti, kako se izboljšave obnašajo glede na globino iskanja, zato sem meritve s posameznimi izboljšavami in brez njih ponovil še za globini šest in tri (vmesne sem izpustil, ker so zanimive le globine, ki so večkratniki števila kart v vzetku). Rezultati so zbrani v tabeli 8.

Algoritem	**Globina 3**		**Globina 6**		**Globina 9**	
	Vozlišča	**Čas (s)**	**Vozlišča**	**Čas (s)**	**Vozlišča**	**Čas (s)**
transpozicijska tabela	162	0,004	5.485	0,142	95.575	2,451
zgodovinska hevristika	97	0,002	6.940	0,129	324.633	5,463
najmanjše okno	266	0,005	15.087	0,276	320.792	4,438
izločanje nekaterih vej	136	0,003	6.879	0,134	258.060	4,891
nič izboljšav	224	0,004	23.892	0,423	1.598.924	21,266

Tabela 8: Učinkovitost izboljšav algoritma alfa-beta glede na globino iskanja

Pri manjših globinah transpozicijska tabela izgubi prednost, ki jo je imela pri večjih. Verjetno je razlog to, da se zaradi manj obiskanih vozlišč vanjo zapiše manj vrednosti, če je bolj prazna, pa tudi ne more biti tako učinkovita. Zgodovinska hevristika, ki je pri večjih globinah malenkost zaostajala za drugimi izboljšavami, se pri manjših globinah najbolj izkaže. To bi se dalo pojasniti s tem, da iskanje pri manjši globini zajame manj različna stanja igre, kar pomeni, da je bolj verjetno, da je poteza, ki je dobra v enem, dobra tudi v drugem. Ne znam pa razložiti, zakaj najmanjše okno pri manjših globinah deluje opazno slabše. Grafično so ti rezultati prikazani na sliki 13. Goli algoritem alfa-beta je zaradi večje preglednosti izpuščen, skala pa je logaritemska.



Slika 13: Učinkovitost izboljšav algoritma alfa-beta glede na globino iskanja

4.4.2. MNENJE IGRALCEV

Ker je program za igranje taroka navsezadnje namenjen predvsem igranju proti človeškim nasprotnikom, je treba ugotoviti, kaj oni menijo o njem. Moj prvi cilj je bil ugotoviti, ali igra dovolj dobro, da je igra proti njemu zanimiva. Poleg tega pa sem želel dognati, ali pri svoji igri uporablja prijeme, kakršne uporabljamo ljudje, a v program niso eksplicitno vnešeni (npr. 'šmiranje', 'tarokiranje' itd.) – torej ali je zmožen odkriti takšne taktike.

Pri tem sta mi pomagala dva dobra tarokista. Program smo pognali z dvema človeškima igralcema in enim računalniškim. Jaz sem bil prvi človek, izmed ostalih preizkuševalcev pa je bil eden drugi človek, drugi pa je spremljal računalnikovo igro. Med seboj si nismo gledali v karte, tako da je bilo mogoče dokaj nepristransko analizirati računalnikovo igro proti dvema človekoma.

Splošen vtis obeh preizkuševalcev je bil precej dober: Silicijastega tarokista sta ocenila za spodobnega, čeprav ne ravno vrhunskega igralca. Proti njemu je zanimivo igrati, ker igra dokaj dobro in do neke mere nepredvidljivo. Zelo očitno slabih potez naredi zelo malo, se pa včasih zdi, da igra brezciljno.

'Šmira' (meče veliko vredne karte, kadar je videti, da bo pobral njegov soigralec) vedno, kadar se ve, da bo soigralec pobral. Kadar pa je na potezi še nasprotnik, se včasih ne odloči pravilno. Te odločitve so pač odvisne od vzorca Monte Carlo in kadar gre za golo verjetnost, je

računalnikova ocena praviloma dobra. Ljudje pa včasih 'šmiramo' tudi, če vemo, da je verjetnost za uspeh majhna, ker lahko predvidimo, da bo kasneje kvečjemu slabša. Silicijasti tarokist tega razen v končnici ne more, ker je njegovo iskanje preplitvo.

'Tarokira' (s svojimi taroki ali malo vrednimi kartami v barvah, ki jih nasprotniki nimajo, druge prisili, da rabijo taroke) včasih, vendar je to njegova šibka točka. 'Tarokiranje' je namreč navadno bolj dolgoročen načrt, kot ga je Silicijasti tarokist zmožen narediti. Vseeno pa mu včasih uspe nenačrtno: ker nima na voljo nobenih posebej perspektivnih potez, vrže tarok, saj je malo vreden in pri njem vsaj izguba ne more biti velika. Je pa tako ravnanje občasno tudi slabost, ker se zgodi, da vrže visok tarok, saj pričakuje, da bo z njim pobral. To se sicer res zgodi, a če tarokov nima veliko, bi ga bilo navadno bolje porabiti za kaj drugega.

Opazili smo tudi nekaj potez, ki bi jih pri človeku označili za zvite. Prvi primer je, da sta igralca pred Silicijastim tarokistom vrgla nizki barvni karti, program pa je pobral s kavalom, čeprav je imel kralja. Naslednji vzetek je potem pobral s tem kraljem. Enemu od preizkuševalcev se je to sicer zdelo preveč tvegano, drugi pa je dejal, da bi sam ravnal enako, in v našem primeru se je izkazalo za pravilno. Drugi primer je, da je igralec odvrgel nizko barvno karto, ker je imel v tisti barvi damo in je upal, da bo s tem izvabil kralja, kar bi mu omogočilo, da bi z damo kasneje pobral. A program je kralja prihranil. Sicer dame ni dobil, ker je imel njen lastnik še eno nizko karto iste barve, vendar se nam je poteza vseeno zdela vredna omembe.

Nekoliko pa je program razočaral pri klopu. Kadar je odvrgel prvo karto vzetka, se je sicer praviloma odločil primerno: navadno je vrgel srednje visoko barvno karto, s katero ni pobral, je pa prisilil nasprotnika, da sta pobrala z najvišjimi kartami. Kadar pa je odvrgel drugo karto vzetka, se je kar nekajkrat odločil za previsoko, tako da je pobral, čeprav bi bil lahko dosegel, da bi pobral zadnji nasprotnik. Tudi pri tretji karti vzetka se je včasih odločil za višjo karto, kot bi bilo treba.

5. SKLEP

V prvem delu naloge sem naredil pregled raziskav na področju računalniškega igranja iger s kartami. Na žalost jih ni prav veliko. Za bridž, ki je deležen še največ pozornosti, obstaja sicer vsaj 10 dobrih programov, vendar o delovanju večine ni dosti znanega. Razlog je verjetno ta, da je navada razlagati delovanje svojih izdelkov doma predvsem v akademskih krogih, ti programi pa so povečini komercialni ali ljubiteljski izdelki. Sam sem se odločil posvetiti taroku in videti je, da sta oba programa zanj, ki sem ju našel, ljubiteljska izdelka, tako da je moj Silicijasti tarokist prvi primer nekoliko bolj znanstvene obravnave računalniškega igranja te igre.

V svojem programu se računalniškega igranja igre s kartami lotim na dokaj običajen način: drevo igre preiskujem s precej izboljšano različico algoritma alfa-beta, nepopolno informacijo pa obravnavam z vzorčenjem Monte Carlo.

Moj preiskovalni algoritem je precej izpopolnjen, težko pa je oceniti, kako blizu je najboljšemu, kar se na tem področju da doseči. Po rezultatih v [7] recimo algoritem alfa-beta s transpozicijsko tabelo, z zgodovinsko hevristiko in z iskanjem z najmanjšim oknom pri iskanju do globine šest porabi približno 35% časa golega alfa-beta. Moje iskanje z istimi izboljšavami pa traja 15% časa iskanja z golim alfa-beta. To se vsekakor zdi spodbudno, vendar so rezultati iskanja po različnih drevesih slabo primerljivi, ker so odvisni od lastnosti drevesa, ocenjevalne funkcije in še česa. Rezultatov za tarok pa nisem našel.

Brez dvoma bi se moje iskanje dalo še izboljšati. Particijsko iskanje je ena pot, čeprav je težko napovedati, kako bi se obneslo. Razen pri bridžu namreč nisem zasledil, da bi bilo kje uporabljeno, pa še pri bridžu sem gotov le, da ga uporablja GIB. Bolj obetavna pot se mi zdi dodatno izločanje vej na podlagi človeškega znanja. To sicer že uporabljam, a v bolj skromnem obsegu, prepričan pa sem, da bi se – posebej ob pomoči dobrega tarokista – dalo še precej dopolniti. Tudi ocenjevalna funkcija, ki je v prvem delu igre, ko preiskovanje ne doseže velike globine, zelo pomembna, bi si zaslužila še nekaj pozornosti (posebej pri klopu in beraču) – tudi tu najbrž s skrbno uporabo človeškega znanja o taroku.

Ugotavljam torej, da preiskovanje drevesa igre – čeprav koristno orodje – ni dovolj. Tarok je preveč kompleksna igra, da bi se dal obvladati z grobo računsko silo. V dober program je treba prenesti tudi človeško znanje o igri. Ali pa iznajti kak povsem nov način.

Vzorčenje Monte Carlo uporabljam v precej osnovni različici. Edino, s čimer poizkušam odpravljati njegove pomanjkljivosti, je dodatno plitvejše iskanje, s katerim dam poudarek takojšnjemu dobičku. To je moja zamisel, ki sem jo uvedel v odgovor na težave, kakršne sem najpogosteje opažal med igro. GIB, ki je danes v samem vrhu programov za igranje bridža, je tudi začel z navadnim vzorčenjem Monte Carlo, čeprav drži, da zdaj uporablja izboljšano različico. Vendar avtor poroča, da je izboljšanje pri igranju ni posebej veliko. [13] Program za igranje pokra Poki pa denimo uporablja selektivno vzorčenje, a ga je pri pokru morda laže implementirati kot pri kaki drugi igri.

Glede na to, da je ugotovljeno, da vzorčenje Monte Carlo ima pomanjkljivosti, bi prav gotovo tudi mojemu programu koristilo, če bi uporabljal kako metodo, ki jih odpravlja. Vendar pa te pomanjkljivosti v praksi niso posebej opazne. Bolj pomembna izboljšava bi bila selektivno vzorčenje. Moji vzorci so namreč v prid večji hitrosti nekoliko premajhni, kar včasih povzroči nedosledno igro in odločitve, ki so očitno posledica prevelikega optimizma ali pesimizma.

Vseeno pa moram skleniti, da je vzorčenje Monte Carlo preprosta in elegantna metoda, ki dosega solidne rezultate. Morebitne izboljšave bi gotovo pripomogle k boljši igri, a prav bistveno ne.

Tudi sama implementacija bi se dala precej pohitriti. Je pa res, da si za hitrost programa niti nisem prizadeval, ker se mi je zdelo koristneje sprogramirati ga tako, da je pregleden in ga je lahko popravljati. Sploh pa ni bil namen naloge izpiliti implementacijo, temveč preizkusiti metode za računalniško igranje iger s kartami.

Zalo zaželena bi bila primerjava Silicijastega tarokista s kakim drugim programom za igranje taroka. Avtorju Taroka_MC sem to sicer predlagal, a je žal ponudbo odklonil.

Kot je pokazal preizkus igranja proti ljudem, je Silicijasti tarokist nasprotnik, ki za igralce je izziv. Ne igra sicer na ravni posebej dobrih tarokistov, je pa dovolj dober, da je igra proti njemu zanimiva. Kljub nekaterim pomanjkljivostim včasih preseneti z nadpovprečnimi potezami. Izboljšati bi bilo treba le igranje klopa.

Silicijasti tarokist je javnosti na voljo šele zelo kratek čas, a če se bo pokazalo, da zanimanje zanj je, bom razvoj najbrž nadaljeval. Tako se bom lahko prepričal, kako dobre so zamisli o izboljšavah, ki sem jih omenjal v prejšnjih odstavkih.

6. ZAHVALA

K moji diplomski nalogi so koristno pripomogli (našteti po abecedi):

Ivan Bratko – ker je imel posluh za moje želje po ukvarjanju z računalniškim igranjem iger, za umne nasvete in ker pač brez njega ne bi šlo;

Mojca Luštrek – ker je uperila svoje ostro lektorsko oko v moj izdelek;

Miran Mlakar – za pomoč pri ocenjevanju programa;

Boštjan Poglajen – ker je Silicijastemu tarokistu naklonil prostor v spletu in za pomoč pri ocenjevanju programa;

Dorian Šuc – za pregled tega besedila.

Za pomoč se jim prav lepo zahvaljujem.

7. LITERATURA

[1] C. Shannon. Programming a Computer for Playing Chess. Philosophical Magazine 41, 1950

[2] A. Turing. Digital Computers Applied to Games. B. Bowden (urednik), Faster Than Thought, Pitman, 1953

[3] A. Samuel. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, 1959

[4] J. Schaeffer. The Games Computers (and People) Play. Advances in Computers 50, 2000

[5] P. Y. Chan, H. Y. Choi, Z. Chiao. Data Structures and Algorithms Class Notes: Game trees, Alpha-beta search. http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11, 1997

[6] G. P. Ingargiola. UGAI Workshop Lectures: MiniMax with Alpha-Beta Cutoff. http://yoda.cis.temple.edu:8080/UGAIWWW/lectures96/search/minimax/alpha-beta.html, 1996

[7] T. A. Marsland. A Review of Game-Tree Pruning. Journal of Computer Chess Association 9(1), 1986

[8] M. L. Ginsberg. Partition Search. AAAI National Conference, 1996

[9] R. Feldmann. Game Tree Search on Massively Parallel Systems. Advances in Computer Chess 7, 1993

[10] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989

[11] Frank & D. Basin. A Theoretical and Empirical Investigation of Search in Imperfect Information Games. Theoretical Computer Science, 1999

[12] J. R. S. Blair, D. Mutchler, M. van Lent. Perfect Recall and Pruning in Games with Imperfect Information. Computational Intelligence, 1996

[13] M. L. Ginsberg. GIB: Imperfect Information in Computationally Challenging Game. Journal of Artificial Intelligence Research 14, 2001

[14] S. J. J. Smith, D. S. Nau. Strategic Planning for Imperfect-Information Games. Games: Planning and Learning Fall Symposium, 1993

[15] K. Erol, J. Hendler, D. S. Nau. HTN Planning: Complexity and Expressivity. AAAI National Conference, 1994

[16] S. J. J. Smith, D. S. Nau, T. Throop. Computer Bridge: A Big Win for AI Planning. AI Magazine 19(2), 1998

[17] D. Billings, A. Davidson, J. Schaeffer, D. Szafron. The Challenge of Poker. Artificial Intelligence, 2001

[18] D. Billings, L. Pena, J. Schaeffer, D. Szafron. Using Probabilistic Knowledge and Simulation to Play Poker. AAAI National Conference, 1999

[19] Spletna stran Bridge Barona. http://www.bridgebaron.com

[20] Spletna stran GIBa. http://www.gibware.com

[21] Spletna stran Taroka. http://members.lycos.co.uk/tarok

[22] Spletna stran Taroka_MC. http://www.tarok.net

[23] Spletna stran Tarok zveze Slovenije. http://users.volja.net/tarokzveza

[24] I. Kononenko. Načrtovanje podatkovnih struktur in algoritmov. Fakulteta za računalništvo in informatiko, 1996

8. IZJAVA O SAMOSTOJNOSTI DELA

Izjavljam, da sem diplomsko nalogo izdelal samostojno pod mentorstvom prof. dr. Ivana Bratka. Sodelavce, ki so mi pri delu pomagali, sem navedel v zahvali.

Mitja Luštrek