RAČUNALNIŠKO IGRANJE IGER

Seminarska naloga pri
metodah umetne inteligence

Mitja Luštrek

2003-09-18

KAZALO

KAZALO 2
1. UVOD 3
2. PREISKOVANJE DREVESA IGRE 4
2.1. MINIMAKS IN ALFA-BETA 4
2.2. B* 5
2.3. DOKAZNA ŠTEVILA 10
2.4. DRUGI PREISKOVALNI ALGORITMI 11
2.4.1. SSS* 11
2.4.2. MTD(f) 12
2.4.3. ZAROTNIŠKA ŠTEVILA 12
2.4.4. BAYESOVSKO PREISKOVANJE DREVESA IGRE 12
2.4.5. APROKSIMACIJA MIN/MAKS 13
2.5. IZBOLJŠAVE 13
2.5.1. TRANSPOZICIJSKA TABELA 13
2.5.2. RAZVRŠČANJE POTEZ 14
2.5.3. PRILAGAJANJE ŠIRINE ISKALNEGA OKNA 15
2.5.4. PRILAGAJANJE GLOBINE ISKANJA 15
2.5.5. PRENAŠANJE VREDNOSTI PO DREVESU IGRE 16
2.5.6. IZBIRA VOZLIŠČA ZA RAZVIJANJE 16
3. ZNANJE 17
3.1. UČENJE IZ RAZLIK V ČASU 17
4. NEPOPOLNA INFORMACIJA 18
4.1. VZORČENJE 18
4.1.1. TEŽAVE VZORČENJA 18
4.1.2. REŠEVANJE TEŽAV VZORČENJA 20
5. KONKRETNE IGRE 21
5.1. BACKGAMMON 21
5.2. BRIDŽ 21
5.3. DAMA 21
5.4. OTHELLO 22
5.5. POKER 22
5.6. SCRABBLE 23
5.7. ŠAH 23
5.8. GO 24
6. EKSPERIMENTALNI REZULTATI 25
6.1. TRIOOMPH 25
6.2. TAROK 26
7. LITERATURA 29

I. UVOD

Računalniško igranje iger ima dokaj dolgo zgodovino. Že v 50. letih 20. stoletja so se tega področja lotili Claude Shannon [40], Alan Turing [49] in morda še posebej Arthur Samuel s svojim programom za damo [36] (čeprav se je ukvarjal predvsem s strojnim učenjem, ki pri igranju iger dandanes ne igra prav velike vloge). Na začetku računalniki nikakor niso bili kos ljudem, danes pa so pri nekaterih igrah že bistveno boljši. K temu sta pripomogla tako bliskovit razvoj strojne opreme, kot tudi boljše razumevanje samih iger ter množica novih algoritmov in prijemov. Gonilna sila razvoja je predvsem šah, ki je bržkone najbolj ugledna in dognana igra ter tudi edina, katere računalniško igranje včasih pritegne pozornost širše javnosti – zmaga, ki jo je Deep Blue [22] slavil nad svetovnim prvakom v šahu Garijem Kasparovom, močno odmevala. Nekateri so celo trdili, da je stroj v tej igri dokončno prevladal nad človekom, vendar je zgolj šest iger, ki sta jih odigrala, premalo za tako sodbo. Pa tudi ljudje se učimo od računalnikov in poznavalci računalniškega šaha lahko svoje znanje izrabijo za učinkovito igro proti šahovskim programom – saj navsezadnje tudi šahovski mojstri študirajo igro svojih tekmecev in se ji prilagajajo. In seveda velja omeniti računalnike tudi kot orodje za preučevanje iger: npr. pri dami so ljudje za 100-letni položaj potrebovali celo stoletje, da so 'dokazali' zmago belega, potlej pa je program Chinook [37] v nekaj sekundah ugotovil, da vodi v remi (položaj je bil takrat prekrščen v 197-letnega). [26]

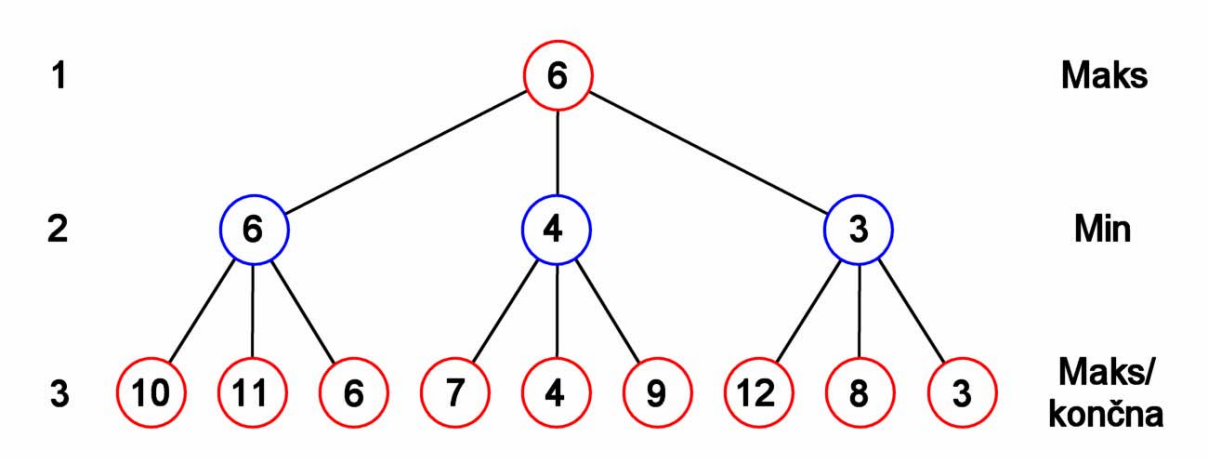
Glavnina raziskav računalniškega igranja iger se ukvarja s preiskovanjem drevesa igre, zato je temu posvečeno 2. poglavje. Tu je največ govora o algoritmu alfa-beta in njegovih izboljšavah, opisanih pa je tudi nekaj alternativ. 3. poglavje se ukvarja z znanjem: načeloma naj bi pravila igre zadoščala, da bi jo računalnik lahko igral, a v resnici to navadno ne omogoča dobrega igranja (oziroma je vsaj mogoče doseči boljše, če programu malo pomagamo). 4. poglavje obravnava igre z nepopolno informacijo ali naključnimi dogodki (npr. metom kocke). 5. poglavje pa govori o nekaj konkretnih igrah in načinih, kako jih računalniki igrajo. In za konec je v 6. poglavju predstavljenih nekaj eksperimentalnih rezultatov uporabe opisanih tehnik pri *trioomphu* in taroku.

2. PREISKOVANJE DREVESA IGRE

Igranje večine iger 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*). [28] 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).

2.1. MINIMAKS IN ALFA-BETA

Ker je v večini iger celotno drevo preveliko, da bi ga razvili do konca, ga razvijemo le do izbrane globine, liste pa hevristično ocenimo – ocenjevalna funkcija je odvisna od problema, s katerim se ukvarjamo. Nato uporabimo algoritem minimaks [13]: če privzamemo, da si prizadevamo za čim večji rezultat, tista vozlišča, kjer smo na potezi mi, označimo z *maks* in v njih izberemo sina z največjim rezultatom; v vozliščih *min* je na potezi nasprotnik in v njih izberemo sina z najmanjšim rezultatom. To ponazarja slika 1. Izraza *maks* in *min* pogosto uporabljamo tudi zase in za nasprotnika, ne le za vrste vozlišč.



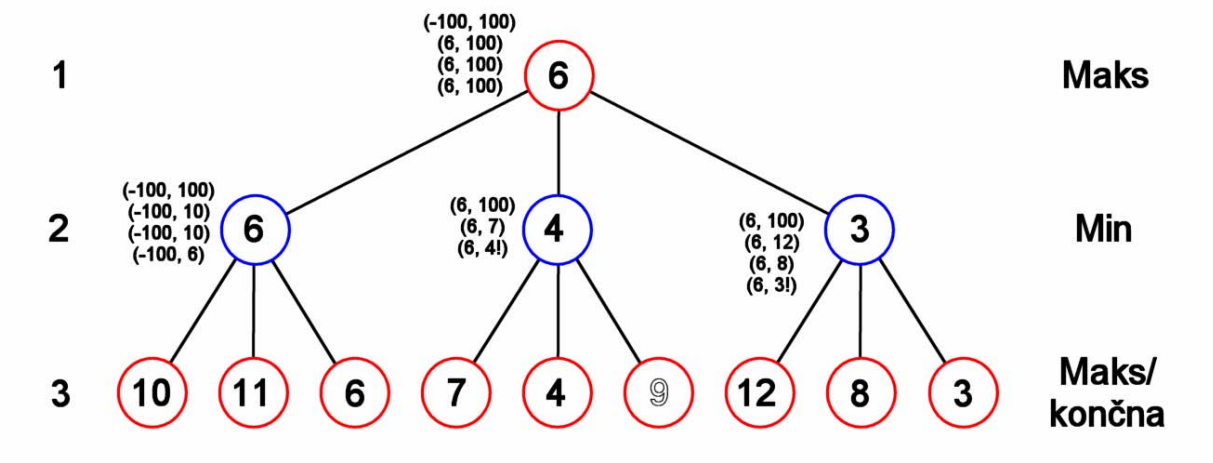
Slika 1. Drevo igre pri algoritmu minimaks

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. To lastnost izkorišča preiskovanje alfa-beta [24], ki je gotovo najbolj razširjen algoritem za igranje iger – to je algoritem 1.

Preiskovanje alfa-beta so v 70. letih 20. stoletja iznašli Donald E. Knuth in sodelavci. Ime 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.

```
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



Slika 2. Drevo igre pri algoritmu alfa-beta

Omeniti velja še **negamaks**, različico algoritma alfa-beta, kjer se plasti min in maks obravnavajo enako, se pa ob vsakem rekurzivnem klicu spremenljivki α, β zamenjata in negirata. Rezultat je enak kot pri običajnem alfa-beta, implementacija pa je bolj kompaktna in manj razumljiva.

2.2. B*

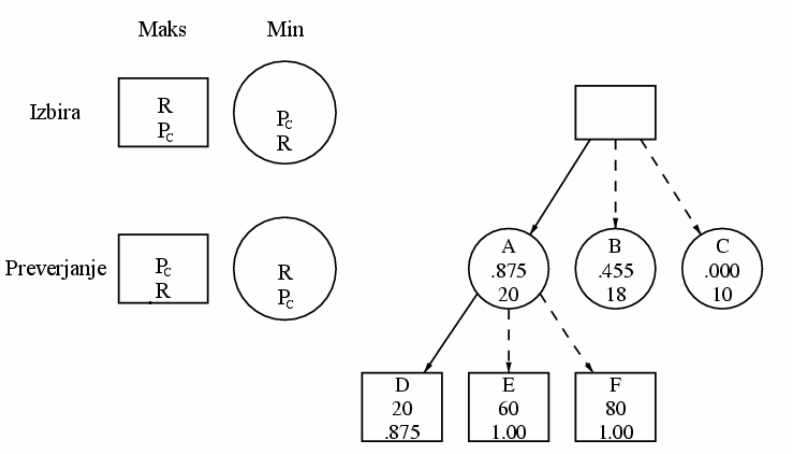
Ta algoritem je razvil Hans M. Berliner leta 1979 [6], zrelo podobo pa je dobil leta 1995 [5]. Temelji na ideji, da ni potrebno točno ugotoviti, kako dobra je vsaka poteza, ki je v danem položaju na voljo, ampak je treba le najti potezo, za katero smo prepričani, da je boljša od vseh drugih. To se najbolj učinkovito stori tako, da se preiskovanje osredotoči na mestih, za katera je najbolj verjetno, da bodo pripeljala do takega prepričanja.

```
B* (koren)
  izbira
    dokler R_najboljši sin korena < O_drugi najboljši sin korena
      cilj := (R_najboljši sin korenu + O_drugi najboljši sin korena) / 2
      za sinove korena izračunaj P_C
      če ima samo en sin korena P_C > P_C min ali čas potekel
        S_naj := najboljši sin korena
        končaj izbiro
      potuj po drevesu navzdol od korena do lista L začenši z maks
        v plasteh maks izbirajoč veje z največjo P_C
        v plasteh min izbirajoč veje z najmanjšo R
      za sinove L izračunaj R
      če je L vrste maks
        za sinove L izračunaj O
      prenesi P, R, O in P_C po drevesu navzgor
  preverjanje
    cilj := R_drugi najboljši sin korena - 1
    za vozlišča vrste min izračunaj O
    prenesi manjkajoče vrednosti po drevesu navzdol in navzgor
    dokler R_najboljši sin korena >= cilj
      za sinove S_naj izračunaj P_C
      če ima samo en sin korena P_C > P_C min ali čas potekel
        končaj B*
      potuj po drevesu navzdol od S_naj do lista L začenši z min
        v plasteh min izbirajoč veje z največjo P_C
        v plasteh maks izbirajoč veje z največjo R
      za sinove L izračunaj R
      če je L vrste min
        za sinove L izračunaj O
      prenesi P, R, O in P_C po drevesu navzgor
  pojdi na izbiro
```

Algoritem 2. B*

B* za vozlišča računa realistično oceno R (to je najboljša ocena, ki jo trenutno lahko ponudi), optimistično oceno O (to je ocena, do katere pride, če je *maks* dvakrat zapored na potezi), pesimistično oceno P (to je optimistična ocena za *min*) in verjetnost P_C (to je verjetnost, da bo pri preiskovanju v tem vozlišču moč doseči cilj – o slednjem več v naslednjem odstavku). Algoritem je sestavljen iz dveh faz: izbire (*select*), kjer *maks* uveljavlja svoj optimizem, in preverjanja (*verify*), kjer svoj optimizem uveljavlja *min*. Če se v drugi fazi za potezo, ki je bila izbrana v prvi, izkaže, da utegne biti slabša, kot je bilo ocenjeno, se prva faza ponovi. To se nadaljuje, dokler preverjanje ne potrdi izbire ali dokler ne zmanjka časa. Do realističnih in pesimističnih ocen B* pride s plitvim preiskovanjem alfa-beta. Podrobneje potek kaže algoritem 2.

Slika 3 kaže začetek primera poganjanja B*. Najprej je na vrsti izbira. *Maks* ima na voljo poteze A, B in C. Za vsako se s preiskovanjem določita R in O. Nato se izračunata cilj, ki je enak ($R_{\text{najboljši sin korena}} + O_{\text{drugi najboljši sin korena}}$) / 2, in P_C, ki je enaka (cilj – R) / (O – R). Najobetavnejša je poteza A (to pomeni, da ima najboljšo možnost, da bo njena ocena dosegla cilj), zato se razvije najprej. *Min* ima na voljo odgovore D, E in F, ki O in P podedujejo od starša (le da se njuni vlogi zamenjata), R in P_C pa se na novo izračunata. Vse te vrednosti kaže tabela 1.

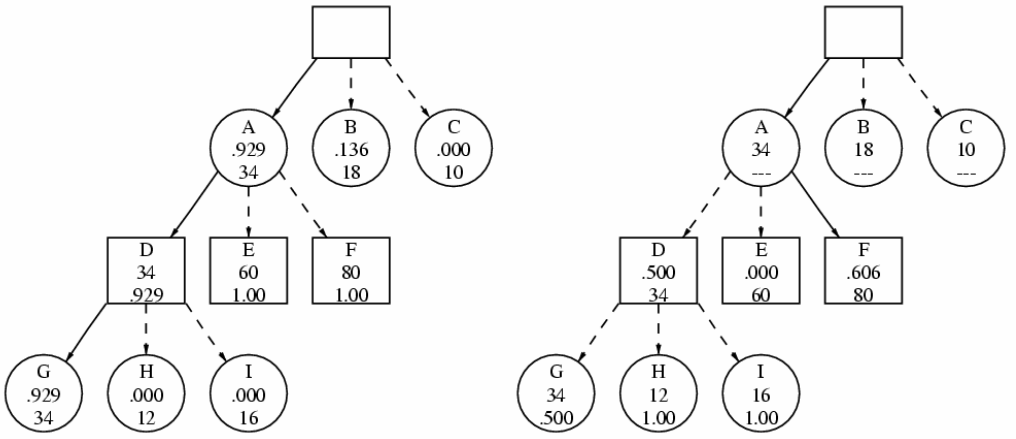


Slika 3. B*: začetek in legenda

Vozlišče	P	R	O	Cilj	P_C
A		20	100	30	0,875
B		18	40	30	0,455
C		10	25	30	0
D	100	20		30	0,875
E	100	60		30	1
F	100	80		30	1

Tabela 1. B*: začetek

Vrednosti P_C v sinovih vozlišča A se med seboj zmnožijo (kajti na sinove vozlišč *min* je treba gledati kot na konjunkcijo; pri sinovih vozlišč *maks* se uporabi največja vrednost) in rezultat se prenese v starša (kar pa v tem primeru ne spremeni njegove vrednosti). Zatem se preiskovanje nadaljuje v vozlišču D, ki je najboljše z vidika *min*. To kaže leva stran slike 4. Vanj se iz najboljšega sina (G) preneseta vrednosti R in O (le da ima O iz vozlišča G vlogo P). Popravijo se tudi vrednosti za vozlišče A, kar spremeni cilj. To se vidi v tabeli 2. R vozlišča A je sicer še vedno malo pod O vozlišča B, tako da povsem ne moremo izključiti možnosti, da bi bila poteza B boljša. A da ne izgubljamo časa, uvedemo spremenljivko, ki določa najmanjšo razliko, ko jo še upoštevamo – manj časa kot nam ostaja za preiskovanje, večja je ta spremenljivka, a tudi na začetku je dovolj velika, da na tej točki začnemo preverjanje (primerna vrednost je 0,15).



Slika 4. B*: izbiranje in preverjanje

Vozlišče	P	R	O	Cilj	P_C
A		34	76	37	0,929
B		18	40	37	0,136
C		10	25	37	0
D	76	34		37	0,929
E	100	60		37	1
F	100	80		37	1
G		34	76	37	0,929
H		12	36	37	0
I		16	32	37	0

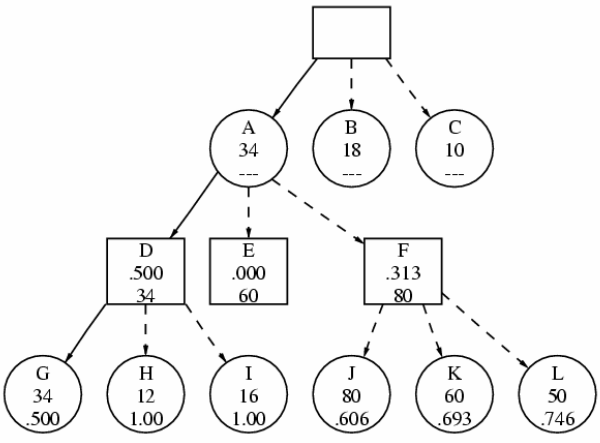
Tabela 2. B*: pred preverjanjem

Na začetku preverjanja se izračunajo O za vsa vozlišča *min* in se prenesejo po drevesu navzdol in navzgor. Cilj se nastavi na $R_{\text{drugi najboljši sin korena}} - 1$, ker si prizadevamo pokazati, da do sedaj najboljša poteza pravzaprav ni najboljša. Na enak način kot med preverjanjem se izračunajo tudi nove P_C. Stanje na tej točki se vidi na levi strani slike 4 in v tabeli 3.

Vozlišče	P	R	O	Cilj	P_C
A	-80	34	76	17	
B		18	40	17	
C		10	25	17	
D	76	34	0	17	0,500
E	100	60	40	17	0
F	100	80	-80	17	0,606
G	0	34	76	17	0,500
H	0	12	36	17	1
I	0	16	32	17	1

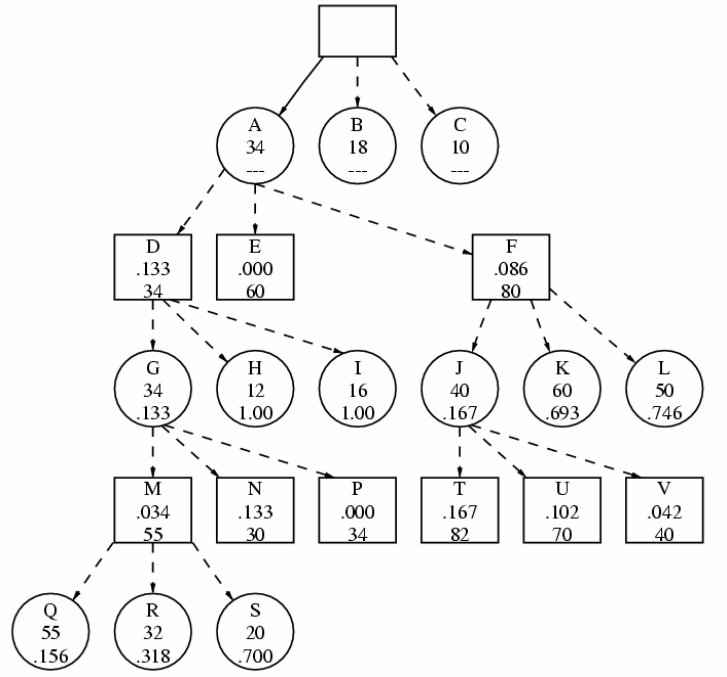
Tabela 3. B*: začetek preverjanja

Izmed sinov vozlišča A ima največjo P_C vozlišče F, zato ga razvijemo najprej – rezultat je na sliki 5. Sinovi vozlišča F podedujejo O in P od starša, R in P_C pa na novo izračunamo. Produkt njihovih P_C (pri preverjanju gre to ravno obratno kot pri izbiri, kjer produkt uporabimo v plasteh *min*, v plasteh *maks* pa najboljšo vrednost) postane P_C vozlišča F.



Slika 5. B*: prvi korak preverjanja

Naslednje vozlišče, ki se ga lotimo, je G – do njega pridemo prek A (*maks*, največja R), D (*min*, največja P_C) in G (*maks*, največja R). Podobno razvijemo še vozlišči M in J, kar nam da končno sliko 6 in tabelo 4. Na tej točki imajo vsi nasledniki A dovolj majhno P_C, da verjamemo, da je A res najboljša poteza.



Slika 6. B*: končno stanje

Vozlišče	P	R	O	Cilj	P_C
A	-80	34	76	17	
B		18	40	17	
C		10	25	17	
D	76	34	0	17	0,133
E	100	60	40	17	0
F	100	80	-80	17	0,086
G	10	34	76	17	0,133
H	0	12	36	17	1
I	0	16	32	17	1
J	4	40	100	17	0,167
K	-80	60	100	17	0,693
L	-80	50	100	17	0,746
M	76	55	10	17	0,034
N	76	30	15	17	0,133
P	76	34	20	17	0
Q	10	55	76	17	0,156
R	10	32	76	17	0,318
S	10	20	76	17	0,700
T	82	82	4	17	0,167
U	80	70	11	17	0,102
V	80	40	16	17	0,042

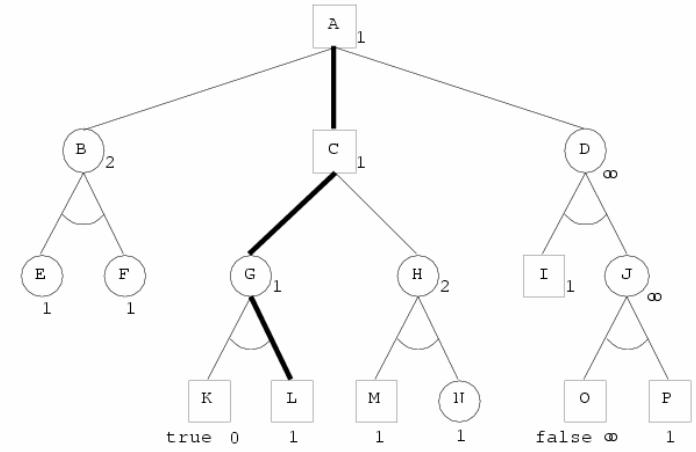
Tabela 4. B*: končno stanje

B* je bil uporabljen pri šahu in se je izkazal za nekoliko slabšega od dodelane različice alfa-beta. Izkazalo pa se je tudi, da hitrejši procesor B* pomaga bolj kot alfa-beta, kar kaže, da bi utegnil v prihodnosti, ko bo na voljo hitrejša strojna oprema, B* preseči alfa-beta. [5]

2.3. DOKAZNA ŠTEVILA

Iskanje z dokaznimi števili (*proof-numbers search*) je iznašel Victor L. Allis leta 1994. [2] V splošnem je to algoritem za preiskovanje dreves in/ali, pri igranju iger pa vozlišča *in* ustrezajo *min* in vozlišča *ali maks*, true pomeni zmago in false poraz. Za igre, kjer moramo obravnavati tudi vmesne vrednosti, iskanje z dokaznimi števili ni najbolj primerno – remi denimo še lahko vključimo, če ga izenačimo z zmago ali porazom, kaj več bi šlo pa teže.

Vsak list drevesa ima vrednost true, false ali neznano. Notranje vozlišče *in* ima vrednost true, če imajo vsi njegovi sinovi vrednost true, false, če ima vsaj en sin vrednost false, in neznano sicer. Notranje vozlišče *ali* ima vrednost true, če ima vsaj en njegov sin vrednost true, false, če imajo vsi sinovi vrednost false, in neznano sicer. Cilj je določiti vrednost korena. Pri tem si pomagamo z dokaznimi (*proof*) in ovržbenimi (*disproof*) števili: to so števila, ki jih priredimo vozliščem in ki povedo, koliko listom poddrevesa s korenom v tistem vozlišču se mora vrednost spremeniti iz neznano v true oziroma false, da to vozlišče dobi vrednost true oziroma false. Dokazna števila za vozlišča *in* so enaka vsoti dokaznih števil sinov, za vozlišča *ali* pa minimumu dokaznih števil sinov. Za ovržbena števila velja ravno obratno. Primer drevesa in/ali z dokaznimi števili je na sliki 7.



Slika 7. Drevo in/ali z dokaznimi števili in potjo, ki jo uberemo pri dokazovanju

Če želimo dokazati ali ovreči koren drevesa, je to najlaže storiti, če začnemo v korenu in potujemo proti listom izbirajoč veje z najmanjšimi dokaznimi ali ovržbenimi števili, ker bomo na ta način morali dokazati ali ovreči najmanj vozlišč. Na sliki 7 je ta pot označena z debelo črto: za dokaz korena moramo dokazati vozlišče L. Ob tem se zastavi vprašanje, ali naj koren poizkušamo dokazati ali ovreči. K sreči pa je presek najmanjše množice listov, ki jih moramo dokazati za dokaz korena, in najmanjše množice listov, ki jih moramo ovreči za ovržbo korena, vedno neprazen, tako da izberemo list iz njega – najbolj dokazujoče vozlišče (*most-proving node*). Tako v vsakem koraku algoritma poiščemo najbolj dokazujoče vozlišče in ga razvijemo, kar ponavljamo, dokler nam na zmanjka časa ali vozlišč – to kaže algoritem 3. Če koren dokažemo, izberemo zmagovito potezo. Če koren ovržemo, izberemo potezo z največjim poddrevesom v upanju, da nasprotnik ne bo opazil, da ima možnost zmagati. Če

nam zmanjka časa, kar je pri mnogih igrah običajno, pa izberemo potezo z najmanjšim razmerjem dokazno število / ovržbeno število.

```
Dokazna števila (koren)
  razvij koren
  določi dokazna in ovržbena števila korena
  dokler (koren.dokazno > 0) in (koren.ovržbeno > 0) in (čas ni potekel)
    V = koren
    dokler je V razvito vozlišče
      če je V vrste ali
        V' := prvi tak naslednik V, da V'.dokazno = V.dokazno
      sicer
        V' := prvi tak naslednik V, da V'.ovržbeno = V.ovržbeno
      V := V'
    razvij V
    določi dokazna in ovržbena števila V
    popravi dokazna in ovržbena števila prednikov V
  če koren.dokazno = 0
    za vse naslednike N korena
      če N.dokazno = 0
        vrni N
  sicer
    če koren.ovržbeno = 0
      N_naj = prvi naslednik korena
      za ostale naslednike N korena
        če velikost poddrevesa N > velikost poddrevesa N_naj
          N_naj = N
      vrni N_naj
    sicer
      N_naj = prvi naslednik korena
      za ostale naslednike N korena
        če N.dokazno / N.ovržbeno > N_naj.dokazno / N_naj.ovržbeno
          N_naj = N
      vrni N_naj
```

Algoritem 3. Iskanje z dokaznimi števili

Iskanje z dokaznimi števili je bilo uporabljeno pri avariju in se je izkazalo za precej boljše od alfa-beta [2] (čeprav danes to ni več relevantno, ker je bil avari leta 2002 rešen [35]).

2.4. DRUGI PREISKOVALNI ALGORITMI

V tem razdelku so opisani še nekateri algoritmi za preiskovanje drevesa igre, za katere ocenjujem, da so manj pomembni ali zanimivi od prejšnjih treh. Spričo velikega števila tovrstnih algoritmov pa je izčrpen pregled malone nemogoče narediti.

2.4.1. SSS*

Ta algoritem je razvil George C. Stockman leta 1979. [42] Namesto preiskovanja v globino, kar uporablja denimo alfa-beta, najprej razvije najobetavnejše vozlišče, zaradi česar bi bilo pričakovati, da se bo obnesel bolje. Problem pa je, da mora vzdrževati dolg sortiran seznam vozlišč, kar zahteva veliko časa in pomnilnika, tako da ni nič hitrejši od alfa-beta, čeprav razvije precej manj vozlišč (da jih nikdar ne razvije več, je celo dokazano). Ker je povrh tudi precej težko razumljiv, se ni dosti uporabljal. Leta 1996 je bil razvit algoritem **MT-SSS*** [32],

ki je zgolj posebna različica alfa-beta s transpozicijsko tabelo, razvije pa ista vozlišča kot SSS* v enakem vrstnem redu. Tudi ta deluje približno enako dobro kot alfa-beta. [32]

2.4.2. MTD(f)

Družino algoritmov MTD so iznašli Aske Plaat in sodelavci leta 1996. [32] V podrazdelku 2.4.1 omenjeni MT-SSS* spada vanjo. Pri teh algoritmih iterativno kličemo alfa-beta z minimalnim iskalnim oknom (γ-1, γ), s čimer postopoma izboljšujemo oceno za zgornjo in spodnjo mejo vrednosti vozlišča. Med klici lahko γ spreminjamo na različne načine: pri MTD(f) uporabimo rezultat prejšnje iteracije. Preizkusi z več igrami kažejo, da je MTD(f) za ~10% hitrejši od dodelane različice alfa-beta. [32]

2.4.3. ZAROTNIŠKA ŠTEVILA

Preiskovanje z zarotniškimi števili (*conspiracy numbers*) je razvil David McAllester leta 1988. [29] Algoritem je soroden dokaznim številom iz razdelka 2.3. Vozliščem priredimo zarotniška števila – to so števila listov v poddrevesu s korenom v tem vozlišču, ki morajo spremeniti svojo vrednost, da se spremeni vrednost vozlišča (oziroma število listov, ki se morajo zarotiti, da bodo spremenili vrednost korena). Najprej razvijemo vozlišča, za katera je bolj verjetno, da bodo povzročila spremembo vrednosti korena; algoritem teče, dokler ni vrednosti korena dovolj zanesljiva ali pa zmanjka časa. Obstaja tudi **alfa-beta-zarotniško preiskovanje** (*alpha-beta-conspiracy search*) [30], ki skuša združiti alfa-beta z idejo zarotniških števil, in **nadzorovano preiskovanje z zarotniškimi števili** (*controlled conspiracy-number search*) [25], ki odpravlja eno izmed težav izvirnega algoritma – nebrzdan razvoj katere izmed vej drevesa igre. Nobena izmed različic pa ni doživela vidnejše uporabe.

2.4.4. BAYESOVSKO PREISKOVANJE DREVESA IGRE

To metodo sta razvila Eric E. Baum in Warren D. Smith leta 1997. [4] Uporablja ocenjevalno funkcijo, ki poleg vrednosti vrne tudi oceno napake, ki jo vrednost ima – ocenjevati napako se nauči tako, da primerja vrednosti ocenjevalne funkcije v nekem vozlišču z vrednostmi pridobljenimi s preiskovanjem drevesa s korenom v tem vozlišču. Na podlagi te ocenjevalne funkcije lahko izračunamo zanesljivost ocene v danem trenutku in tudi koliko bi se zanesljivosti ocene povečala, če bi razvili vse liste trenutnega drevesa igre – to vrednost označimo z U. Nadalje za vsak list izračunamo, koliko se poveča U, če bi ga razvili. List z največjo absolutno vrednostjo te spremembe nato dejansko razvijemo, kar se ponavlja, dokler na zmanjka časa. Absolutno vrednost upoštevamo zato, ker je zmanjšanje U koristno (pomeni, da je odločitev bliže), drastično povečanje pa sumljivo (pomeni, da imamo napačno predstavo o situaciji). Tovrstno preiskovanje drevesa igre je bilo preizkušeno v programu za *Othello* in izkazalo se je, da je primerljiv z Eclipsom [33], v tistem času drugim najboljšim programom na Internet Othello Serverju (ki ne obstaja več, ima pa naslednika [19]). [4]

2.4.5. APROKSIMACIJA MIN/MAKS

Aproksimacijo min/maks (*min/max approximation*) je razvil Ronald L. Rivest leta 1988. [34] Uporablja posplošeno p-povprečje n vrednosti vektorja a:

$$M_p(a) = (\frac{1}{n}\sum_{i=1}^{n} a_i^p)^{1/p}$$

M_1 (a) je navadno povprečje, M_∞ (a) je maksimum in $M_{-\infty}$ (a) je minimum; izberemo primeren p (npr. 10), nato pa v plasteh *maks* uporabimo M_p in v plasteh *min* M_{-p}. Iz teh dveh funkcij lahko izračunamo, koliko vsak naslednik vpliva na vrednost starša. Te vplive priredimo povezavam v drevesu kot kazni – večji vpliv da manjšo kazen. Nato izberemo list z najmanjšo kaznijo in ga razvijemo. To ponavljamo, dokler ne zmanjka časa. Metoda je bila v igri štiri v vrsto primerjana z alfa-beta in izkazalo se je, da je pri enaki časovni omejitvi alfa-beta nekoliko boljši, aproksimacija min/maks pa pregleda bistveno manj vozlišč (vendar za vsakega porabi več časa). [34]

2.5. IZBOLJŠAVE

Algoritmi opisani v prejšnjih štirih razdelkih se dajo na mnoge načine izboljšati ali vsaj prirediti. Prijemi opisani v tem razdelku se načeloma nanašajo na alfa-beta, so pa nekateri uporabni tudi drugje.

2.5.1. 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. Ta tabela se imenuje transpozicijska (*transposition table*) [28] 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.

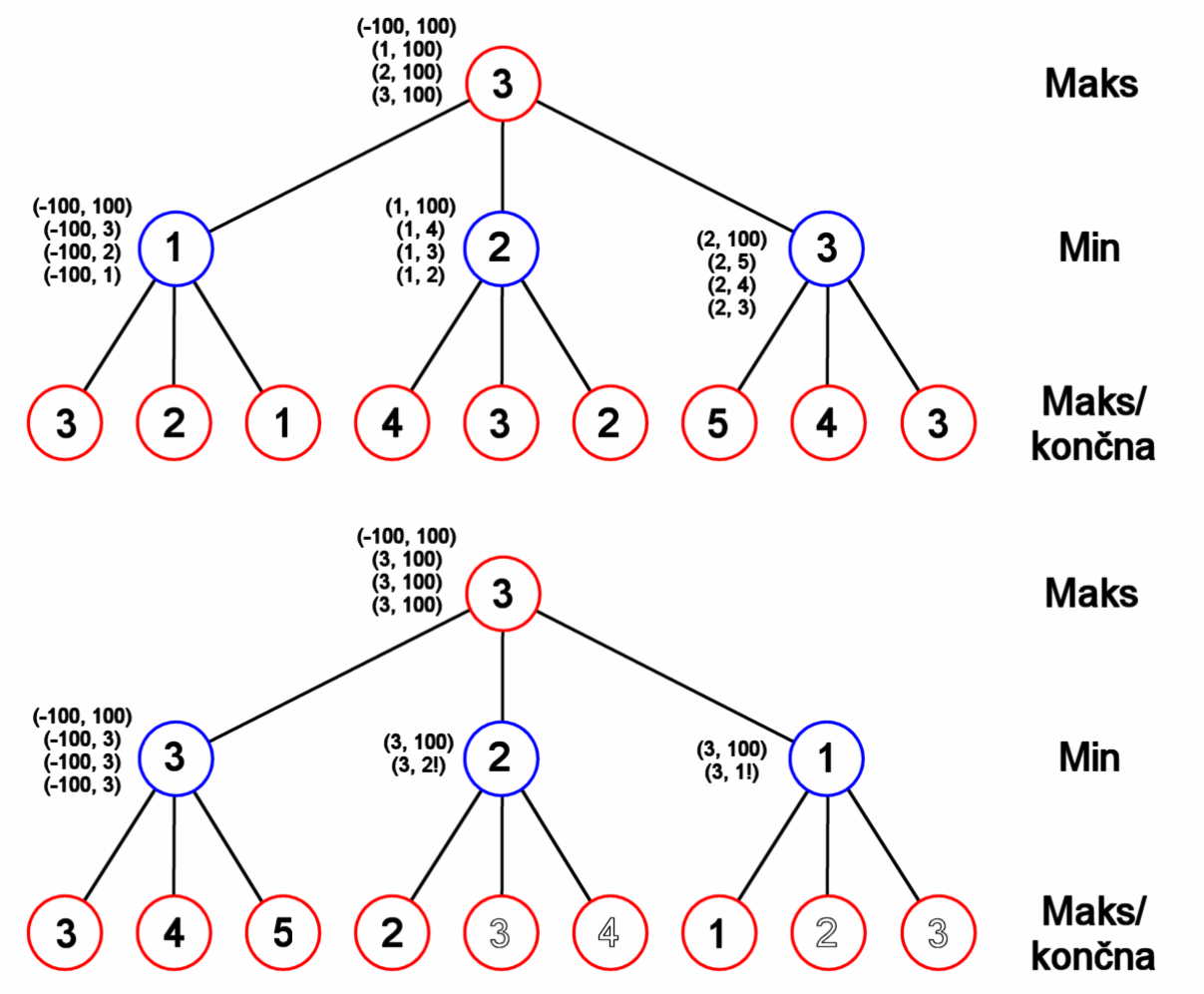
Če je takrat, ko smo stanje shranili, algoritem alfa-beta napravil rez, shranjene vrednosti ne moremo uporabiti kot točno vrednost, ker zaradi reza ne vemo, kakšna točna vrednost je, lahko pa ga uporabimo za prilagoditev vrednosti spremenljivke α ali β. V tabeli za vsako stanje lahko označimo, katera poteza je v njem povzročila rez (če ga kaka je), in kadar stanje najdemo v tabeli (ne moremo pa shranjene vrednosti uporabiti kot točno vrednost), tisto potezo preiščemo najprej, ker ga bo najbrž spet. Posebno uporabno je to v kombinaciji z iterativnim poglabljanjem. Iterativno poglabljanje pomeni, da preiskovanje kličemo večkrat z naraščajočo globino. Zapisi iz prejšnjih iteracij tako lahko usmerjajo iskanje v kasnejših.

Mnogo stanj, ki so shranjena v transpozicijski tabeli, si je zelo podobnih, zato bi jih bilo zaželeno združiti. To izkorišča **particijsko iskanje** (*partition search*), ki ga uporablja trenutno najbrž najboljši program za bridž GIB. [20] Za vsako obiskano stanje poišče sosednja stanja, ki imajo enako vrednost ocenjevalne funkcije, in jih v transpozicijsko tabelo shrani kot množico. Podobna deluje **ekvivalenčna transpozicijska tabela** (*equivalence transposition table*), ki se uporablja v programu za tarok Silicijasti tarokist. [27] Razlika je v tem, da se pri Silicijastemu tarokistu stanja združujejo hevristično in ni zagotovila, da imajo povsem enako vrednost, lahko pa se jih na ta način združi več.

Pri različnih igrah je učinek transpozicijske tabele različno velik: pri šahu je pospešitev lahko 75%, pri dami 89%, pri *Othellu* pa le 33% [38]; pri taroku je z navadno transpozicijsko tabelo 86% in z ekvivalenčno do 99%.

2.5.2. RAZVRŠČANJE POTEZ

Iskanje se močno pospeši, če v vsakem vozlišču najprej preiščemo najboljšega naslednika. Meja iskanja pri algoritmu alfa-beta se v tem primeru že po prvem nasledniku nastavi na končno vrednost za trenutno vozlišče in pri neprvih vozliščih hitro pride do rezov. Slika 8 kaže koristnost razvrščanja – vozlišča v zgornjem drevesu so razvrščena slabo (rezov sploh ni), v spodnjem pa dobro (rezi so povsod, kjer je možno).



Slika 8. Razvrščanje potez

Razvrščanje potez lahko časovno zahtevnost algoritma alfa-beta z O (vejitev^{globina}) zmanjša na O (vejitev$^{globina\ /\ 2}$). Pri šahovskih programih se prva poteza preišče najprej v do 90% primerih, pri *Othellu* pa v do 80% primerih. [38]

Kot je omenjeno v prejšnjem podrazdelku, lahko kot metoda za razvrščanje potez služi transpozicijska tabela, posebej v kombinaciji z iterativnim poglabljanjem. Ker utegne biti pri nekaterih iskanjih transpozicijska tabela premajhna, tako da se veliko položajev prepiše, se uporablja tudi **ovržbena tabela** (*refutation table*) [28]. Ta je manjša in shranjuje le poteze, ki so v preteklosti povzročile reze.

Podobna metoda je **ubijalska hevristika** (*killer heuristic*) [17]. 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*) [39]. 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 vedno na voljo metode, ki temeljijo na znanju o problemu, s katerim se ukvarjamo.

2.5.3. 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*) [38]. Izid iskanja bo eden izmed tehle:

$n - \delta <$ rezultat $< n + \delta$: iskanje je bilo krajše, kot bi bilo z navadnim algoritmom 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 window search*, tudi *principal variation search*) [28]. 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).

2.5.4. PRILAGAJANJE GLOBINE ISKANJA

Ker so nekatere poteze bolj zanimive od drugih, jih je zaželeno preiskati globlje.

Iskanje z upoštevanjem stabilnih stanj (*quiescence search*) [17] plitveje preišče stabilna (*quiet*) stanja, ker ta stabilnost pomeni, da od njih ne vodi dosti zanimivih možnosti in se zato lahko ocenijo z malo nadaljnjega iskanja. Ta metoda je zelo pogosta pri šahu.

Iskanje z ničelno potezo (*null-move search*) [31] izvede plitvejše iskanje, kjer ima ena stran dve zaporedni potezi. Če niti na ta način ne more doseči česa omembe vrednega, potem tam globlje iskanje ni potrebno. Ta prijem je načeloma koristen, je pa nevaren v položajih, kjer je narediti potezo škodljivo (položaji *zugzwang* pri šahu).

Edinstvene poglobitve (*singular extensions*) [3] so metoda, ki s spreminjanjem širine iskalnega okna ugotovi, ali je v nekem položaju najboljša poteza bistveno boljša od druge

najboljše. To naj bi pomenilo, da se na tistem mestu dogaja nekaj zanimivega, zato se preiskovanje poglobi.

ProbCut [12] uporablja plitvejše iskanje za ugotavljanje, ali bo globlje dalo zanimiv rezultat. Povezava med plitvimi in globokimi iskanji se dožene na podlagi statistične analize iskanj. Ta metoda je uspešno uporabljena v trenutno najbrž najboljšem programu za *Othello* Logistellu [11], pri npr. šahu pa ne deluje zaradi prešibke povezave med rezultati globokih in plitvih iskanj.

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

2.5.5. PRENAŠANJE VREDNOSTI PO DREVESU IGRE

Običajno se vrednosti po drevesu navzgor prenašajo tako, da se v plasteh *min* vzame minimum vrednosti naslednikov vozlišča, v plasteh *maks* pa maksimum. V idealnih okoliščinah bi to zadoščalo, ker pa navadno ne poznamo točnih vrednosti listov, se je razvilo več alternativ [23].

Prenašanje M in N (*M&N backup*) namesto ene same vrednosti v starša prenese v plasteh *min* N in v plasteh *maks* M najboljših vrednosti. To odraža dejstvo, da je bolje imeti več dobrih možnosti kot eno samo. Problem pa je, da ne deluje z algoritmom alfa-beta, ker slednji izračuna točno vrednost samo za najboljšega naslednika vozlišča.

Pravilo prenašanja s produktom (*product-propagation rule*) v vozliščih *maks* verjetnost zmage izračuna kot $1 - \Pi (1 - p_i)$, v vozliščih *min* pa $1 - \Pi p_i$; p_i so verjetnosti zmage v naslednikih vozlišča. Tudi ta metoda zahteva točne vrednosti vseh naslednikov.

Povprečno prenašanje (*average propagation*) uporabi povprečje rezultata minimaksa in pravila prenašanja s produktom.

2.5.6. IZBIRA VOZLIŠČA ZA RAZVIJANJE

Pri preiskovanju drevesa igre se uporabljajo različni vrstni redi obiskovanja vozlišč: od preprostega preiskovanja v globino pri alfa-beta, do izbiranja najboljšega vozlišča po raznoterih kriterijih. Problem metod iz druge skupine je, da navadno zahtevajo hranjenje dobršnega dela drevesa igre v pomnilniku. Tule naj omenim še dve metodi za izbiro naslednjega vozlišča, ki ga bomo razvili [23].

Ekvipotencialno iskanje (*equi-potential search*) za vsak list hevristično oceni verjetnost, da bo njegovo razvitje spremenilo najboljšo potezo (S), in čas, ki bi ga to razvitje vzelo (P). Nato iterativno preiskuje drevo igre in v vsaki iteraciji razvije liste, ki imajo količnik P/S manjši od praga E; E se z iteracijami povečuje. Ta metoda je pravzaprav oblika preiskovanja v globino, tako da ni potratna s pomnilnikom.

Minimaks najboljši najprej (*best-first minimax*) vedno razvije vozlišče na koncu trenutno najboljšega zaporedja potez. Zanaša se na to, da je vrednost ocenjevalne funkcije po *maksovi* potezi navadno večja kot po *minovi*, kar zagotavlja, da ne podaljšuje ves čas istega zaporedja potez.

3. ZNANJE

Najbolj osnovna uporaba znanja je **transpozicijska tabela**. Čeprav se navadno obravnava kot nekaj začasnega, bi načeloma lahko stanja in njihove vrednosti shranili za uporabo v kasnejših igrah – to se počne, čeprav redko.

Drug pogost način uporabe znanja je **baza končnic**. Za enostavne končne položaje je moč določiti ali npr. vodijo v zmago, poraz ali remi in to uporabljati namesto ocenjevalne funkcije, ko preiskovanje seže dovolj globoko. Za šah obstajajo baze vseh položajev s petimi figurami, za damo pa celo z osmimi. [38]

Knjižnice otvoritev hranijo zaporedja začetnih potez, ki izvirajo iz literature in iger, ki so jih odigrali mojstri – lahko tudi mojstrski programi.

Znanje o igri pa seveda vsebuje tudi **ocenjevalna funkcija**. Navadno je to utežena vsota lastnosti položaja (*features*). Te lastnosti so večinoma določene ročno, uteži pa včasih ročno, včasih pa tudi samodejno.

3.1. UČENJE IZ RAZLIK V ČASU

Učenje iz razlik v času (*temporal difference learning*) je metoda, s katero se je ukvarjal že Arthur Samuel leta 1959 [36], bolj opazna pa je postala, ko jo je Richard S. Sutton leta 1988 formaliziral [43]. Je oblika vzpodbujevalnega učenja (*reinforcement learning*), kjer se program iz končnega rezultata skuša naučiti, katere poteze so v nekem položaju dobre in katere slabe. To stori tako, da določi uteži ocenjevalne funkcije.

Naj ocena položaja pomeni verjetnost končne zmage – imenujmo jo P_t, kjer je t čas oziroma zaporedna poteza. Med igro program naredi zaporedje ocen P_1, P_2, ..., P_N. Če naj bo ocenjevalna funkcija uporabna, mora ocena ob času t predvideti oceno ob času t + 1, zato si učenje iz razlik v času prizadeva uteži določiti tako, da bo razlika $P_{t+1} - P_t$ čim manjša. Pri tem upošteva vse poteze do t-te, le da zgodnejšim pripiše eksponentno manjši pomen. Popravki uteži ob času t se računajo po enačbi:

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_w P_k$$

Vektor uteži je označen z w, $\nabla_w P_k$ je vektor parcialnih odvodov P_k po vseh utežeh, λ je parameter, ki določa, kako močan naj bo vpliv na pretekle napovedi ($0 \leq \lambda \leq 1$), α pa parameter, ki določa hitrost učenja ($\alpha > 0$).

To metodo zelo uspešno uporablja trenutno najbrž najboljši program za *backgammon* TD-Gammon [44]. Uporabljena pa je bila med drugim tudi v šahovskem programu Cilkchess [45], kjer je igranje programa izboljšala, vendar avtorji poročajo, da so rezultati težko razumljivi, ker učenje iz razlik v času nekatere lastnosti položaja 'razume' drugače, kot so bile zamišljene. [38]

4. NEPOPOLNA INFORMACIJA

Igre z nepopolno informacijo in igre z naključnimi dogodki (npr. metom kocke) so trši oreh od iger s popolno informacijo. Nepopolno informacijo v igrah si lahko predstavljamo kot množico svetov, od katerih vsak ustreza enemu možnemu stanju igre, igralci pa ne vedo, v katerem svetu so, ker ne vedo, kakšno stanje igre dejansko je. [18] 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. Običajnejša predstavitev drevesa igre z nepopolno informacijo vsebuje dodatne veje za možna stanja igre, vendar se prej opisana v nadaljevanju izkaže za prikladnejšo.

Najti optimalno strategijo za igro z nepopolno informacijo je NP-poln problem. [9] Njegova časovna zahtevnost je namreč eksponentna glede na velikost drevesa igre (pri minimaksu pa je linearna).

4.I. VZORČENJE

Glede na to, da ne moramo obravnavati celotne množice možnih svetov, lahko iz nje izberemo nekaj vzorcev in zanje igro obravnavamo kot igro s popolno informacijo. Tako lahko npr. pri taroku za nekaj možnih razdelitev kart ostalih igralcev preiščemo drevo igre in za vsako najdemo najboljšo potezo; poteza, ki se bo izkazala za najboljšo pri največ razdelitvah, je najbrž tudi res najboljša. Uporabimo lahko vzorčenje Monte Carlo (*Monte Carlo sampling*) [18], kjer so vzorci izbrani naključno, ali selektivno vzorčenje (*selective sampling*) [38], kjer so vzorci reprezentativni. Prednost druge metode je, da ji za enake rezultate zadošča manj vzorcev, sicer pa sta dokaj enakovredni. Izbiro poteze z vzorčenjem prikazuje algoritem 4.

```
Poteza z vzorčenjem
  vzorci := 0
  za vse možne poteze
    ocena [trenutna poteza] := 0
  ponovi n-krat; n je izbrano število vzorcev
    stanje igre := Dopolni z neznano informacijo (znano stanje igre)
    (naj poteza, ocena_naj poteza) := Preišči drevo igre (stanje igre)
    ocena [naj poteza] := ocena [naj poteza] + ocena_naj poteza
  naj poteza := prva poteza
  za vse možne poteze razen prve
    če ocena [trenutna poteza] > ocena [naj poteza]
      naj poteza := trenutna poteza
  vrni naj poteza
```

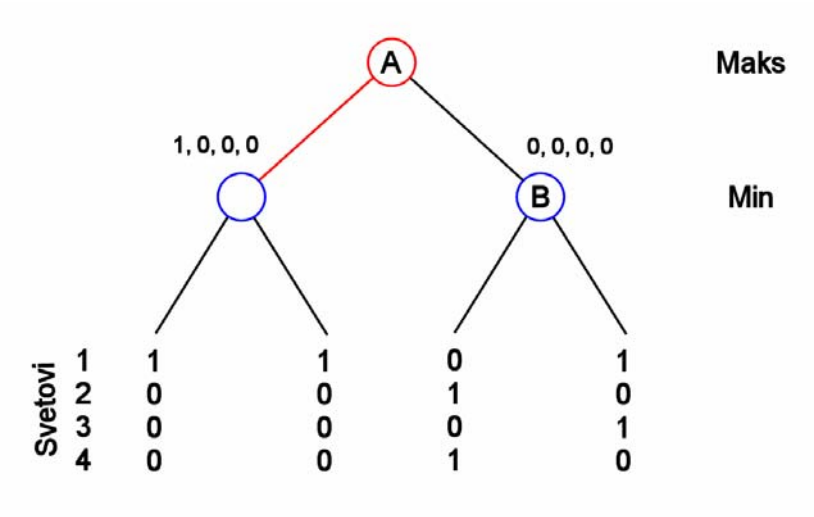
Algoritem 4. Izbira poteze z vzorčenjem

4.I.I. TEŽAVE VZORČENJA

Vzorčenje je malone edini praktičen način za igranje kompleksnih iger z nepopolno informacijo, vendar ima nekaj težav, ki so neodvisne od števila vzorcev (nastopile bi tudi, če bi obravnavali vse svetove). [18]

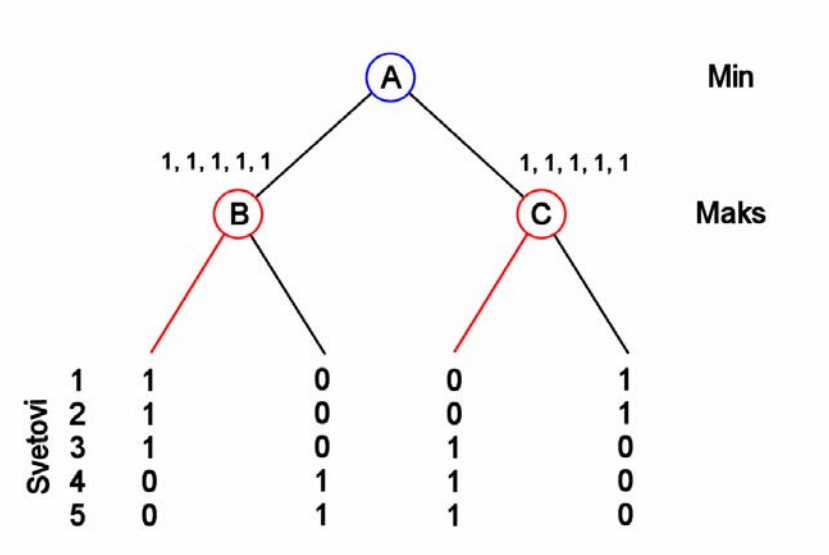
Prva je, da **predpostavlja, da ima *min* popolno informacijo o igri**, zaradi česar ne izkoristi dejstva, da je ponavadi nima. To težavo prikaže slika 9. Na njej *maks* v vozlišču A izbere levo

stran, ker je vektor vrednosti po svetovih (na slikah 9-11 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 kar v polovici primerov izbral napačno in bo *maks* zmagal.



Slika 9. *Maks* napačno predpostavlja, da je *min* popolno informiran

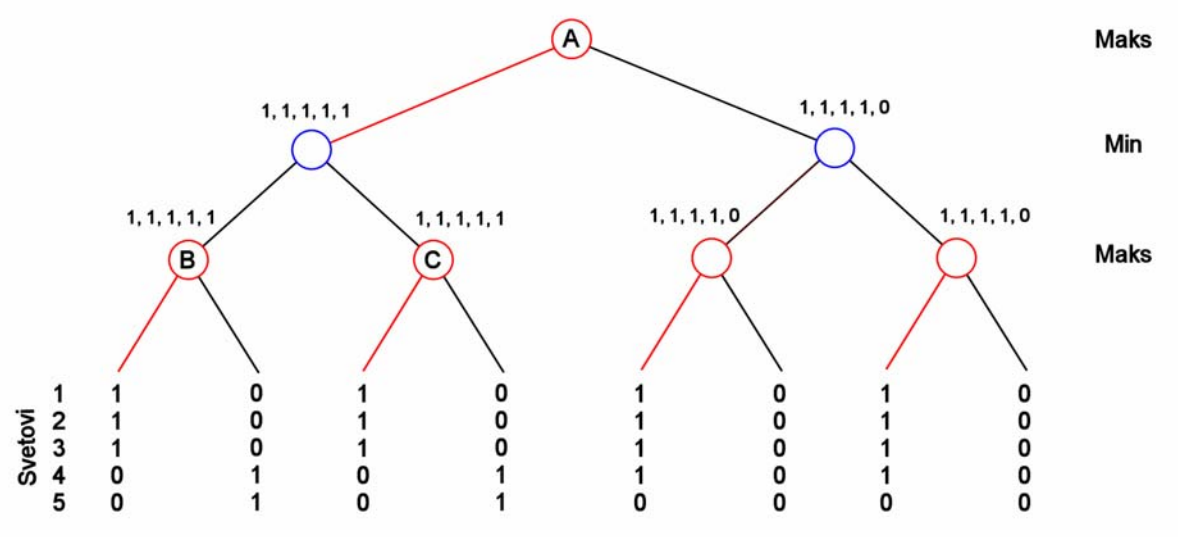
Druga težava je, da se pri vzorčenje ravna, kot da bi bilo odločanje v vsakem vozlišču odvisno le od drevesa pod njim – **preveč lokalno**. V resnici namreč *min*, če ve, kateri svet je pravi in kako igra *maks* (kar je naša predpostavka), lahko izkoristi maksovo nepopolno informiranost. To kaže slika 10. 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 10. *Min* izkoristi *maksovo* neinformiranost

Tretja težava pa je, da vzorčenje **nekatere odločitve prelaga na kasnejše poteze**. Npr. pri taroku 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 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 11 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 11. Maks (napačno) predpostavlja, da bo še izvedel, kateri svet je pravi

4.1.2. REŠEVANJE TEŽAV VZORČENJA

Omenjene tri težave bi bilo koristno rešiti, a praktična rešitev obstaja le za zadnjo. To je **optimizacija s škripajočim kolesom** (*squeaky wheel optimization*), ki jo uporablja trenutno najbrž najboljši program za bridž GIB. [20] Pri njej tvorimo izbrano število vzorcev, nato pa jih dodajamo v množico. Vsakič, ko dodamo vzorec, preverimo, ali imamo načrt igranja, ki deluje za celo množico – ali je množica dosegljiva (*achievable*). Če je, element zares dodamo, sicer pa ne. Ker si želimo, da bi bila ta množica čim večja, postopek ponovimo večkrat in vzorce, ki jih v prejšnjih iteracijah nismo uspeli dodati (so škripali), v naslednjih poizkusimo dodati prej (ko smo še manj omejeni z vzorci, ki so že v množici).

Nekaj podobnega lahko dosežemo z **vektorskim minimaksom** (*vector minimaxing*), težavo s preveliko lokalnostjo pa odpravlja **minimaks z zmanjševanjem vrednosti** (*payoff-reduction minimaxing*). [18] Vendar ti dve metodi zahtevata običajni minimaks (ne alfa-beta), zaradi česar sta prepočasni in zato zanimivi samo teoretično.

5. KONKRETNE IGRE

Obstajajo računalniški programi za ogromno iger, tule pa so opisani le za nekaj najbolj vidnih. Večina napisanega je povzetega po [38], rezultati z računalniške olimpiade leta 2002 pa po [1].

5.1. BACKGAMMON

Prvi resnejši program za *backgammon* je bil BKG9.8 avtorja Hansa Berlinerja, ki je leta 1979 premagal človeškega svetovnega prvaka. Vendar je bil dvoboj kratek in analiza je pokazala, da je program igral slabše, a je imel precej sreče. V 80. letih 20. stoletja je začel svoj program razvijati Gerry Tesauro in leta 1998 se je s človeškim svetovnim prvakom spopadel TD-Gammon 3.0. Človek je tesno zmagal, analiza pa je pokazala, da je z izjemo ene hude napake program igral bolje.

TD-Gammon [44] za ocenjevalno funkcijo uporablja nevronsko mrežo, ki ima za vhod celotno stanje igre (kar nanese približno 300 nevronov), njen izhod pa je verjetnost zmage. Naučena je bila s pomočjo učenja iz razlik v času v približno 1.500.000 igrah programa proti samemu sebi. Uporablja triplastno preiskovanje, pri katerem zaradi vejitve okrog 400 (ki je posledica meta kocke) upošteva le najbolj verjetne nasprotnikove poteze.

Zmagovalec računalniške olimpiade 2002 je BGBlitz [7] (TD-Gammon se je ni udeležil).

5.2. BRIDŽ

Prvi poizkusi na področju računalniškega bridža segajo v 60. leta 20. stoletja. A zgodnji programi so bili zelo slabi (ali kot je izjavil večkratni svetovni prvak Bob Hamman, »treba bi jih bilo izboljšati, da bi bili brezupni«). Leta 1990 je Zia Mahmood, prav tako večkratni svetovni prvak, obljubil 1.000.000 GBP programu, ki bi ga premagal. V 90. letih sta Dana Nau in Steve Smith razvila program Bridge Baron, ki je uporabljal planiranje in ki ni bil več tako slab, a se z vrhunskimi človeškimi igralci še vedno ni mogel primerjati. Od leta 1998 pa je najboljši program za bridž GIB Matthewa L. Ginsberga, ki je Mahmooda celo pripravil do tega, da je umaknil svojo nagrado.

GIB [20] za licitiranje uporablja bazo 7.400 pravil (licitiranje pri bridžu je namreč precej zapleteno, ker služi tudi sporazumevanju med partnerjema) in simulacijo (hitro odigrane cele igre). Pri igranju uporablja particijsko iskanje in vzorčenje Monte Carlo, ki ga dopolnjuje z optimizacijo s škripajočim kolesom.

Zmagovalec računalniške olimpiade 2002 je WBridge5 [50] (GIB se je ni udeležil).

5.3. DAMA

Eden izmed pionirjev umetne inteligence Arthur Samuel se je začel ukvarjati s programom za damo že leta 1948. A njegov glavni cilj ni bilo toliko dobro igranje dame kot narediti program, ki se uči.V 70. letih 20. stoletja je na Duke University nastal program, ki je bil boljši od Samuelovega, a po neuspešnih poizkusih organiziranja dvoboja s človeškim svetovnim prvakom je zanimanje zanj zamrlo. Leta 1989 pa so Jonathan Schaeffer in sodelavci napisali

Chinook, ki je leta 1992 tesno izgubil proti človeškemu svetovnemu prvaku. Dvoboj je bil ponovljen leta 1994, a ga je človek iz zdravstvenih razlogov predal. Chinook je svetovnega prvaka premagal še dvakrat in jasno je, da mu ljudje niso več kos.

Chinook [37] uporablja algoritem alfa-beta z množico izboljšav, ki je povrh še paralelen – leta 1994 je tekel na 18 procesorjih in je dosegal povprečno globino od 19 do 45 plasti (v povprečju pa 25). Ocenjevalna funkcija je določena ročno – avtomatski načini se niso obnesli. Opremljen je tudi z obsežno knjižnico otvoritev in bazo končnic, ki vsebuje vse položaje z do osmimi figurami. Zaradi globokega iskanja se občasno zgodi, da bazo končnic doseže že v prvi potezi.

Zmagovalec računalniške olimpiade 2002 je Dam 2.2 [16] (Chinook se je ni udeležil).

5.4. OTHELLO

Prvi omembe vreden program za igranje *Othella* je Iago avtorja Paula Rosenblooma iz začetka 80. let 20. stoletja, ki je gladko premagoval druge tovrstne programe. Proti vrhunskemu človeškemu igralcu pa je igral le dvakrat in obakrat izgubil. S prestola ga je izrinil Bill, ki sta ga razvila Kai-Fu Lee in Sanjoy Mahajan. Tudi ta proti vrhunskim človeškim igralcem ni dosti igral, edinkrat, ko je, pa je zmagal. Vendar se iz ene igre ne da zanesljivo sklepati o moči programa. V 90. letih se je pojavil Logistello avtorja Michaela Bura, ki je mnogo boljši od Billa, in leta 1997 je igral šest iger proti človeškemu svetovnemu prvaku – vseh šest je dobil.

Logistello [11] za preiskovanje uporablja probCut z iterativnim poglabljanjem, veliko transpozicijsko tabelo in ubijalsko hevristiko. Za zadnjih 22-26 potez lahko igro navadno reši, tako da od tam naprej igra brez 'razmišljanja'. Ocenjevalna funkcija vključuje nabor zanimivih vzorcev, ki se pojavljajo v igri, vrednosti pa so pridobljene iz rezultatov odigranih iger in so lahko različne za vsako od 13 faz igre (ki so določene glede na število igranih ploščic). Program uporablja veliko otvoritveno knjižnico, ki je zgrajena iz iger s tekmovanj, ki so naknadno natančneje analizirane, in iz iger programa proti samemu sebi.

Zmagovalec računalniške olimpiade 1992 (zadnje na kateri se je *Othello* igral) je Othel du Nord.

5.5. POKER

Prvi opaznejši program za poker je Orac poklicnega pokeraša Mika Cara iz leta 1984, ki se je izkazal v nekaj igrah proti dobrim človeškim igralcem, vendar o njegovem delovanju ni dosti znanega, pa tudi rezultatov ni dovolj za zanesljivo sodbo o njegovi kvaliteti. V 90. letih 20. stoletja so se s pokrom pod vodstvom Darsa Billingsa začeli ukvarjati na University of Alberta, rezultat pa sta bila programa Loki in Poki, ki človeškim mojstrom še nista bila kos. Njihov najnovejši dosežek pa je PsOpti, za katerega se zdi, da je že blizu najboljšim ljudem, čeprav ga bo za zanesljivo sodbo treba še temeljiteje preizkusiti.

PsOpti [8] je precej različen od običajnih programov za igranje iger, pri katerih je osnova preiskovanje drevesa igre. Uporablja namreč vnaprej izračunano optimalno strategijo za poenostavljeno igro. Pravo optimalno strategijo je namreč bistveno pretežavno izračunati, zato je množica vseh možnih listov razdeljena na ekvivalenčne razrede (še obvladljivo število je sedem), pa tudi nekaj drugih poenostavitev je uporabljenih. Za tako igro so avtorji PsOptija optimalno strategijo izračunali v slabem dnevu, predvsem pa jih je omejevalo le 2 GB

pomnilnika. Med igranjem program odloči, v kateri ekvivalenčni razred spada njegov list in nato za ustrezno fazo igre iz tabele odčita verjetnosti, da je optimalno višati, izenačiti ali odstopiti (ker gre za igro z nepopolno informacijo, je optimalna strategija določena z verjetnostmi). V skladu s temi verjetnostmi nato igra.

5.6. SCRABBLE

Prvi znan program za *Scrabble* sta napisala Stuart Shapiro in Howard Smith leta 1977. Na računalniških olimpiadah je večkrat zmagal TSP avtorja Jima Homana (ki je kasneje postal komercialen program Crosswise), zelo dober pa je že od leta 1986 Maven, ki se olimpiad ne udeležuje. Leta 1998 je Maven porazil ekipo, ki sta jo sestavljala človeški svetovni prvak in podprvak, tako da je videti, da je pri *Scrabblu* računalnik prekosil človeka.

Scrabble je igra z nepopolno informacijo, zato si Maven avtorja Briana Shepparda pomaga z vzorčenjem in triplastnim preiskovanjem. A kljub plitvemu preiskovanju bi bilo preiskati vse možnosti prezamudno, saj jih je v povprečju okrog 700. Zato uporablja tri generatorje potez – prvi izbira dobre poteze, ki hkrati puščajo dobre možnosti za nadaljnjo igro, drugi izbira poteze, ki nasprotniku preprečujejo, da bi v eni potezi igral vse ploščice in tako dobil dodatne točke, in tretji izbira poteze zgolj po številu točk, ki jih prinašajo. Vsak da približno 10 potez, kar po izločanju enakih da 20-30 možnosti – te se vse upoštevajo. Proti koncu igre, ko je znano, katere ploščice ima nasprotnik, se za preiskovanje uporablja B*.

5.7. ŠAH

Prvi šahovski program je napisal Alan Turing leta 1950, prvi dokumentiran primer delujočega šahovskega programa pa je iz leta 1956. Računalnik (MacHack VI) se je šahovskega tekmovanja (Massachusetskega amaterskega) prvič udeležil leta 1966 in je enkrat remiziral, štirikrat pa izgubil. Prvo tekmovanje v računalniškem šahu je bilo organizirano leta 1970, udeležilo se ga je šest tekmovalcev, zmagal pa je Chess 3.0. Prvi specializiran šahovski računalnik je bil Chess Challenger iz leta 1977. Takrat so računalniki začeli zmagovati tudi proti dobrim šahistom in dosegati spodobne rezultate na boljših tekmovanjih – predvsem so se odlikovali pri hitropoteznem šahu. Najvidnejši uspeh računalniškega šaha je zmaga IBMovega Deep Blue proti svetovnemu prvaku Gariju Kasparovu leta 1997. [15] To sicer ne pomeni dokončne prevlade računalnikov nad ljudmi v šahu, ker je bilo odigranih le šest iger, potlej pa je bil računalnik razstavljen in tako nadaljnjih rezultatov ni pričakovati, je pa vseeno velik uspeh.

Deep Blue [22] je bil 30-procesorski računalnik, vsak procesor pa je nadzoroval še 16 specializiranih šahovskih čipov. Slednji so imeli vgrajen preiskovalni algoritem, generator potez in ocenjevalno funkcijo. Preiskovalni algoritem je bil vzporeden alfa-beta z iterativnim poglabljanjem, transpozicijsko tabelo in edinstvenimi poglobitvami. Uporabljal je bazo končnic s pet ali manj figurami, majhno otvoritveno knjižnico in bazo velemojstrskih iger. Ocenjevalna funkcija je imela čez 8000 parametrov in je bila po večini določena ročno.

Zmagovalec računalniške olimpiade 2002 je Junior [14]; na lestvici The Swedish Chess Computer Association [47], ki primerja šahovske programe, pa je na prvem mestu Shredder 7.04 [14].

5.8. GO

V nasprotju z do sedaj naštetimi igrami igrajo računalniki go precej slabo. Prvi program za go je napisal D. Lefkovitz leta 1960. Šele konec 70. let 20. stoletja pa je se je pojavil prvi program, ki ni zmagoval le proti popolnim začetnikom (avtorja Bruca Wilcoxa). Danes je programov za go kar precej (morda celo več kot za katerokoli drugo igro razen za šah), še vedno pa igrajo na ravni amaterja. Zato ima zasluge veliko igralno polje (19 × 19) in velika svoboda pri vlečenju potez. V goju je tako možnih ~10^{160} položajev (v šahu ~10^{50}), najmanjše drevo igre, s katerim bi se go dalo rešiti, pa bi imelo ~10^{400} vozlišč (za šah ~10^{123}).

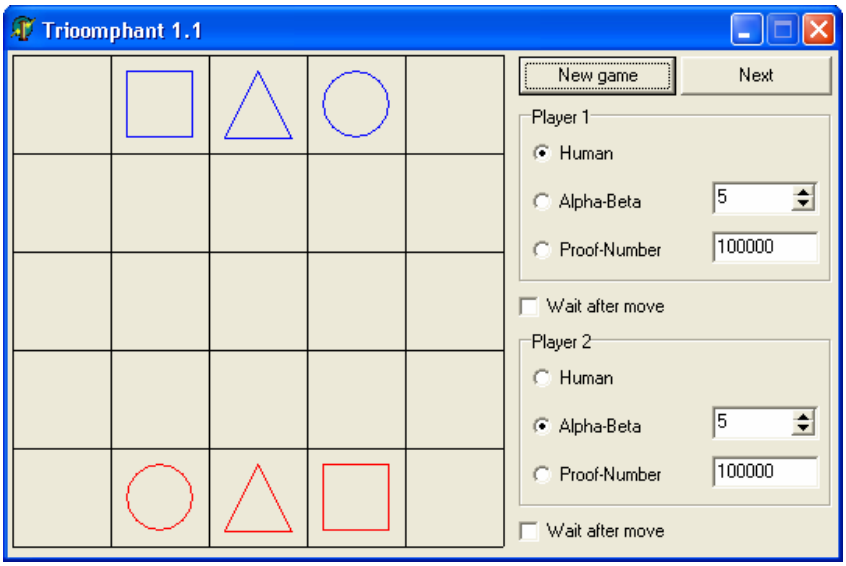
Ker programi za go ne morejo uporabljati globalnega preiskovanja drevesa igre, svoja prizadevanja razdelijo tako prostorsko (se hkrati ukvarjajo le s posamičnimi skupinami kamenčkov) kot tudi glede na cilj (uporabljajo veliko generatorjev potez in ocenjevalnih funkcij, od katerih ima vsak svoj cilj). Pri preiskovanju je iskanje z dokaznimi števili vsaj tako pogosto uporabljano kot alfa-beta – tudi zato, ker se podatki, ki jih prvo hrani v pomnilniku, lahko prenesejo iz enega iskanja v drugo. Iskanje je pogosto dopolnjeno z bazo značilnih vzorcev kamenčkov, ki se poizkusijo prilagoditi med igro pojavljajočim se vzorcem. [10]

Zmagovalec računalniške olimpiade 2002 je Go4++ [21], ki vodi tudi na The Computer Go Ladder [46].

6. EKSPERIMENTALNI REZULTATI

6.1. TRIOOMPH

Trioomph [48] je preprosta igra, na kateri sem primerjal alfa-beta in preiskovanje z dokaznimi števili. Igrata jo dva igralca, ki izmenično premikata figure za eno polje, njun cilj pa je bodisi požreti nasprotnikov kvadrat ali pa obe ostali figuri. Trikotna figura lahko požre kvadratno in kvadratna okroglo, okrogla pa lahko trikotno spremeni v okroglo.



Slika 12. Trioomphant – program za igranje *trioompha*

Moj program Trioomphant 1.1, ki ga je moč videti na sliki 12, lahko igra tako z alfa-beta kot s preiskovanjem z dokaznimi števili, za primerjavo pa sem uporabil Trioomph 1.0 [48] avtorja Žiga Hajdukoviča, ki si je najbrž izmislil tudi samo igro.

Tabela 5 kaže rezultate igre Trioomphanta z alfa-beta proti Trioomphu. 'Globina' pomeni globino preiskovanja, 'Rezultat' rezultat igre (da se programa zaciklata, pomeni, da brez konca ponavljata zaporedje potez, zaradi katerega so krožno izmenjuje nekaj položajev), 'Vozlišča' povprečno število razvitih vozlišč na potezo in 'Čas' povprečen čas potreben za eno potezo izmerjen na računalniku s procesorjem Pentium III 700 MHz in 256 MiB pomnilnika.

Globina	Rezultat	Vozlišča	Čas (s)
1	zaciklata se, Trioomph boljši	175	0,000588
2	Trioomphant zmaga v 12 potezah	1.314	0,0183
3	Trioomphant zmaga v 16 potezah	8.056	0,114
4	Trioomphant zmaga v 7 potezah	31.449	0,514
5	Trioomphant zmaga v 8 potezah	350.057	9,554

Tabela 5. Trioomphant z alfa-beta proti Trioomphu

Tabela 6 kaže rezultate igre Trioomphanta s preiskovanjem z dokaznimi števili proti Trioomphu. 'Vozlišča/možnost' pomeni največje število vozlišč, ki jih program sme razviti za ocenitev ene izmed možnih potez v vsakem položaju.

Vozlišča/možnost	Rezultat	Vozlišča	Čas (s)
10.000	Trioomphant zmaga v 9 potezah	125.991	1,108
20.000	zaciklata se (Trioomph boljši)	239.179	1,921
30.000	Trioomph zmaga v 15 potezah	343.596	2,842
40.000	zaciklata se (Trioomph boljši)	411.943	3,327
50.000	zaciklata se (Trioomph boljši)	553.542	4,720
60.000	Trioomphant zmaga v 8 potezah	661.876	5,884
70.000	Trioomphant zmaga v 8 potezah	769.366	6,910
80.000	Trioomphant zmaga v 5 potezah	1.060.705	9,534
90.000	Trioomphant zmaga v 5 potezah	1.192.704	10,729
100.000	Trioomphant zmaga v 5 potezah	1.324.695	12,027

Tabela 6. Trioomphant s preiskovanjem z dokaznimi števili proti Trioomphu

Zmaga Trioomphanta z najšibkejšo različico preiskovanja z dokaznimi števili je očitno anomalija. Sicer pa je iz točk, kjer se zmagovalca zamenjata, moč sklepati, da sta si alfa-beta globine 1 ali 2 in preiskovanje z dokaznimi števili s 50.000-60.000 vozlišči/možnost približno enakovredna. Neposredna primerjava obeh algoritmov v Trioomphantu pokaže, da pri alfa-beta globine 1 preiskovanje z dokaznimi števili vedno zmaga, sicer pa nikoli.

Glede na počasnost preiskovanja z dokaznimi števili je alfa-beta absoluten zmagovalec. To gre najbrž pripisati predvsem temu, da preiskovanje z dokaznimi števili ne uporablja nobenega znanja o *trioomphu* (razen tega, kateri položaji so zmage in kateri porazi). Sicer preiskovanje z dokaznimi števili ima nekaj prednosti: če je gol algoritem na voljo, ga je moč izredno hitro prilagoditi konkretni igri, za vsako vozlišče porabi zelo malo časa in kadar zmaga, zmaga zelo hitro (kar je najbrž posledica tega, da ne upošteva drugega kot zmage in poraze). Vseeno pa kljub relativni enostavnosti igre deluje slabo; glede na to, da v nekaterih igrah deluje dobro, je to po vsem videzu algoritem, ki je dosti manj splošno uporaben od alfa-beta.

6.2. TAROK

Učinkovitost nekaterih izboljšav algoritma alfa-beta je prikazana na programu za tarok Silicijastem tarokistu [41], ki ga kaže slika 13. Za različne kombinacije izboljšav so prešteta obiskana vozlišča in je izmerjen čas, ki je za preiskovanje potreben na računalniku s procesorjem Athlon XP P1800+ in 512 MiB pomnilnika.

Slika 13. Silicijasti tarokist – program za igranje taroka

Uporabljene izboljšave so iterativno poglabljanje (IP), zgodovinska hevristika (ZH), selektivna tvorba potez (STP), transpozicijska tabela (TT0-TT4: TT0 je navadna, TT1-TT4 pa so ekvivalenčne, pri čemer ima TT1 najmanjše in TT4 največje ekvivalenčne razrede) ter iskanje z najmanjšim oknom (INO). Razen selektivne tvorbe potez so vse metode opisane v podpoglavju 2.5; selektivna tvorba potez pa pomeni, da se namesto vseh dovoljenih potez upoštevajo le tiste, ki niso očitno slabe ali enakovredne drugim.

Tabela 7 kaže učinkovitost različnih kombinacij izboljšav. Rdeči del prikazuje optimalno kombinacijo v celoti (to je IP, ZH, STP in TT3 – izbrana vrsta transpozicijske tabele nudi najboljše razmerje med pravilnostjo in hitrostjo) in brez po ene izboljšave. Rumena je optimalna kombinacija z dodatkom iskanja z najmanjšim oknom, ki pa se ne obnese, čeprav iskanje z najmanjšim oknom samo zase je koristno – to se vidi iz modrega dela, kjer so vključene posamične izboljšave (razen iterativnega poglabljanja, ki samo zase ni koristno), na koncu pa tudi nobena.

Izboljšave	Vozlišča	Čas (s)
IP, ZH, STP, TT3	2.470	0,068
ZH, STP, TT3	3.799	0,119
IP, STP, TT3	3.325	0,091
IP, ZH, TT3	9.936	0,212
IP, ZH, STP	24.034	0.392
IP, ZH, STP, TT3, INO	4.067	0,114
ZH	283.673	3,966
STP	40.838	0,625
TT3	14.457	0,293
INO	605.381	7,120
nobena izboljšava	1.598.924	21,266

Tabela 7. Izboljšave alfa-beta

Vidi se, da najbolj pomaga transpozicijska tabela in nato selektivna tvorba potez, zgodovinska hevristika in iterativno poglabljanje sta manj učinkovita, iskanje z najmanjšim oknom pa lahko sploh ni koristno.

Tabela 8 primerja različice transpozicijske tabele: rdeči del z vključenim iterativnim poglabljanjem in modri brez.

Izboljšave	Vozlišča	Čas (s)
IP, ZH, STP, TT0	7383	0,409
IP, ZH, STP, TT1	6367	0,219
IP, ZH, STP, TT2	3190	0,102
IP, ZH, STP, TT3	2470	0,068
IP, ZH, STP, TT4	1523	0,037
ZH, SPP, TT0	22877	1,299
ZH, SPP, TT1	13044	0,473
ZH, SPP, TT2	8142	0,265
ZH, SPP, TT3	3799	0,119
ZH, SPP, TT4	1023	0,025

Tabela 8. Vrste transpozicijske tabele

Ekvivalenčna transpozicijska tabela 4, ki je sicer najhitrejša, ni uporabna, ker pri igranju povzroča preveč napak. Ostale so sprejemljive, posebej ker se pri taroku uporablja vzorčenje in se zato preiskovanje izvede večkrat, kar vpliv napak nekoliko zmanjša. Primerjava rdečega in modrega dela pokaže, da iterativno poglabljanje preiskovanje lahko močno pospeši, vendar se z uvedbo ekvivalenčne transpozicijske tabele njegov učinek precej zmanjša. Zaradi več zadetkov v tabeli namreč razvrščanje potez, pri katerem iterativno poglabljanje pomaga, ne pride toliko do izraza, dejstvo, da se v vsaki iteraciji del preiskovanja ponovi, pa ostaja.

7. LITERATURA

[1] 7th Computer Olympiad. http://www.cs.unimaas.nl/olympiad2002/
[2] Allis, V. L. Searching for Solutions in Games and Artificial Intelligence. Doktorat, Universiteit Maastricht, 1994
[3] Anatharanam, T., Campbell, M. in Hsu, F. Singular Extensions: Adding Selectivity to Brute-Force Searching. Artificial Intelligence 43(1), 1990
[4] Baum, E. B. in Smith, W. D. A Bayesian Approach to Relevance in Game Playing. Artificial Intelligence 97(1-2), 1997
[5] Berliner, H. J. in McConnel, C. B* Probability Based Search. Artificial Intelligence, 1995
[6] Berliner, H. J. The B* Tree Search Algorithm: A Best-First Proof Procedure. Artificial Intelligence 12(1), 1979
[7] BGBlitz. http://www.bgblitz.com/
[8] Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T. in Szafron, D. Approximating Game-Theoretic Optimal Strategies for Full-scale Poker. Proceedings of IJCAI, 2003
[9] Blair, J. R. S., Mutchler, D. in Lent, M. van. Perfect Recall and Pruning in Games with Imperfect Information. Computational Intelligence, 1996
[10] Bouzy, B. in Cazenave, T. Computer Go: an AI Oriented Survey. Artificial Intelligence 132(1), 2001
[11] Buro, M. Logistello – A Strong Learning Othello Program. 1997
[12] Buro, M. Probcut: An Effective Selective Extension of the α-β Algorithm. Journal of the International Chess Association 18(2), 1995
[13] Chan, P. Y.; Choi, H. Y.; Chiao, Z. Data Structures and Algorithms Class Notes: Game trees, Alpha-beta search. http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic11, 1997
[14] Chessbase.com. http://www.chessbase.com/shop/
[15] Computer Chess History by Bill Wall. http://www.geocities.com/SiliconValley/Lab/7378/comphis.htm
[16] Dam 2.2. http://www.xs4all.nl/~hjetten/dameng.html
[17] Feldmann, R. Game Tree Search on Massively Parallel Systems. Advances in Computer Chess 7, 1993
[18] Frank, I. in Basin, D. A Theoretical and Empirical Investigation of Search in Imperfect Information Games. Theoretical Computer Science, 1999
[19] Generic Game Server. http://external.nj.nec.com/homepages/igord/usrman.htm
[20] Ginsberg, M. L.. GIB: Imperfect Information in Computationally Challenging Game. Journal of Artificial Intelligence Research 14, 2001
[21] Go++. http://www.goplusplus.com/
[22] Hsu, F. IBM's Deep Blue Chess Grandmaster Chips. IEEE Micro, 1999
[23] Junghanns, A. Are There Practical Alternatives to Alpha-Beta in Computer Chess? ICCA Journal 21, 1998
[24] Knuth, D. E. in Moore, R. W. An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(1), 1975
[25] Lorenz, U., Rottman, V., Feldmann, R. in Mysliewietz, P. Controlled Conspiracy-Number Search. ICCA Journal 18(3), 1995
[26] Loy, J. The 197 Years Problem. http://www.jimloy.com/checkers/hundred.htm, 1999
[27] Luštrek, M. Računalniško igranje iger s kartami. Diplomska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2002

[28] Marsland, T. A. A Review of Game-Tree Pruning. Journal of Computer Chess Association 9(1), 1986
[29] McAllester, D. Conspiracy Numbers for Min-Max Search. Artificial Intelligence 35(3), 1988
[30] McAllester, D. in Yuret, D. Alpha-Beta-Conspiracy Search. 1993
[31] Morland, B. Computer Chess. http://www.seanet.com/~brucemo/chess.htm
[32] Plaat, A., Schaeffer, J., Pijls, W. in de Bruin, A. Best-First Fixed-Depth Minimax Algorithms. Artificial Intelligence 87(1-2), 1996
[33] Reversi: The Eclipse. http://www.freeverse.com/flash/eclipse.mgi
[34] Rivest, R. L. Game Tree Searching by Min/Max Approximation. Artificial Intelligence 34(1), 1988
[35] Romein, J. W. in Bal, H. E. Awari is Solved. ICGA Journal 25(3), 2002
[36] Samuel, A. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, 1959
[37] Schaeffer, J. One Jump Ahead: Challenging Human Supremacy in Checkers. Springer Verlag, 1997
[38] Schaeffer, J. The Games Computers (and People) Play. Advances in Computers 50, 2000
[39] Schaeffer, J. The History Heuristic and Alpha-Beta Search Enhancements in Practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989
[40] Shannon, C. Programming a Computer for Playing Chess. Philosophical Magazine 41, 1950
[41] Silicijasti tarokist. http://tarok.bocosoft.com
[42] Stockman, G. C. A Minimax Algorithm Better than Alpha-Beta? Artificial Intelligence 12(2), 1979
[43] Sutton, R. S. Learning to Predict by Methods of Temporal Differences. Machine Learning 3, 1988
[44] Tesauro, G. Temporal Difference Learning and TD-Gammon. Communications of the ACM 38(3), 1995
[45] The Cilkchess Parallel Chess Program. http://supertech.lcs.mit.edu/chess/
[46] The Computer Go Ladder. http://www.cgl.ucsf.edu/go/ladder.html
[47] The Swedish Chess Computer Association. http://w1.859.telia.com/~u85924109/ssdf/
[48] Trioomph. http://zigah.rs-pi.com/trioomph/
[49] Turing, A. Digital Computers Applied to Games. B. Bowden (urednik), Faster Than Thought, Pitman, 1953
[50] WBridge5. http://perso.chello.fr/users/y/yvescostel/