SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET

# BRIŠKULA

SEMINARSKI RAD IZ PREDMETA UMJETNA INTELIGENCIJA

IVAN LAKOVIĆ
MIRJANA JUKIĆ-BRAČULJ

Izv. prof. dr. sc. LUKA GRUBIŠIĆ

Zagreb, 12.1.2016

# 1. Uvod

Briškula ( Briscola ) je talijanska kartaška igra udomaćena na hrvatskom priobalju i otocima. Iznimno je popularna i lako se uči. Najčešće se igra s tršćanskim kartama ( tal. Carte Triestine ). U špilu ima 40 karata. Po istim osnovnim pravilima igra se obična i dupla briškula. Običnu igraju dva igrača ili 4 igrača od kojih svaki ima 3 karte. Dupla briškula se igra u dvoje. Svaki igrač ima 4 karte. Na kraju dijeljenja karata igračima okreće se jedna karta licem prema gore i na nju se stavlja preostali špil karata. Okrenuta karta se zove briškula i ona predstavlja tip ( boju ) karte u koju se igra. Postoje 4 tipa karata: kupe, baštone, špade i dinari. Karte su označene brojevima 1-7 i 11-13. Karte poredane po veličini, od najjače do najslabije su redom: as ( 1 ), trica ( 3 ), kralj ( 13 ), konj ( 12 ), fanta ( 11 ) i dalje od sedmice ( 7 ) prema dvici ( 2 ). As se broji kao 11 punata, trica 10, kralj donosi 4, konj 3, fanta 2, a ostale karte ne donose punte. Igru započinje igrač kojem su prvom podijeljene karte. Bačena karta predstavlja tip karte u koju se igra ta runda. Igrač s najjačom kartom te runde, kupi karte, prvi uzima novu kartu sa špila i započinje novu rundu. Prva bačena karta se može pobijediti jačom kartom istog tipa ili bilo kojom briškulom. U igri ukupno ima 120 punata, pobjednik je onaj tko ima više od 60. U slučaju da obje strane imaju 60 punata tada nema pobjednika. Igra se na 4 dobivene igre.

Igru smo odlučili implementirati u programskom jeziku python koji ima modul za izradu grafičkog sučelja pod nazivom pygame. Čitajući o heuristikama koje se koriste u kartaškim i društvenim igrama odlučili smo implementirati Monte Carlov algoritam i stablo pretraživanja koje se temelji na simulaciji velikog broja igara kako bi se donijela odluka o najboljem potezu. Algoritam ne zahtjeva nikakve dodatne heuristike, a izrazito je popularan u svijetu igara.

## 2. Monte Carlo stablo pretraživanja (engl. Monte Carlo tree search ( MCTS ) )

Monte Carlo stablo pretraživanja je heuristički algoritam pretraživanja koji se koristi u procesima donošenja odluka, najčešće u igranju igara. MCTS algoritam se koncentrira na nalaženje poteza koji najviše obećava, a temelji se na izgradnji stabla pretraživanja na slučajnom prostoru pretraživanja. Igra se simulira veliki broj puta. U svakoj simulaciji igra se igra od trenutnog stanja do kraja. Konačni ishodi simuliranih igara se koriste za određivanje težina čvorova. Čvorovi s većim težinama imaju veći izgled da će biti izabrani u idućim simulacijama. Nakon završenih simulacija odabire se onaj potez koji je donio najviše pobjeda igraču koji je na potezu. Svaka simulacija igre se sastoji od 4 koraka.

1. Odabir: krećući od korijena R, odabiremo najuspješnije čvorove sve do lista L. U nastavku ćemo objasniti odabir čvorova u ovom koraku.
2. Proširivanje: ako list L ne označava kraj igre, stvorimo mu novo dijete C.
3. Simulacija: od čvora C pa na dalje simuliramo igru.
4. Ažuriranje znanja: koristeći rezultat odigrane simulacije, ažuriramo informacije u čvorovima od C do R.

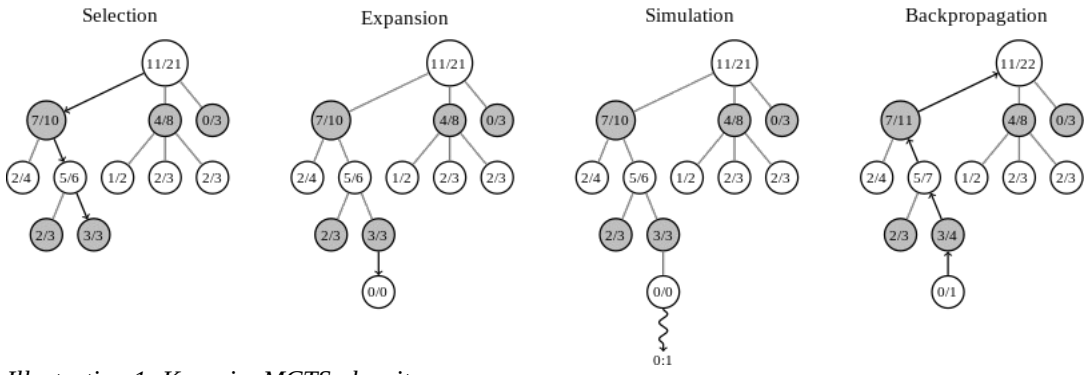


*Illustration 1: Koraci u MCTS algoritmu*

Važno je uočiti da se ažuriranje broja pobjeda u čvorovima vrši iz gledišta igrača koji je napravio taj potez. To nam osigurava da svaki igrač prilikom odabira idućeg poteza teži proširenju poteza koji najviše obećava.  Na kraju se odabire jedan čvor-dijete od korijena R.

Najteži posao je održati ravnotežu između iskorištavanja kombinacija koje donosi odabir djeteta s visokim stupnjem pobjeda i proširivanja poteza s malim brojem simulacija. Prva formula koja je to uspjela zove se UCT ( Upper Confidence Bound 1 applied to trees ). Odabire se onaj potez koji ima najveću vrijednost dobivenu po UCT formuli. Navedena formula glasi:

$\frac{w_i}{n_i} + c\sqrt{\frac{\ln(t)}{n_i}}$ pri čemu:

- $w_i$ označava broj pobjeda nakon i-tog poteza
- $n_i$ označava broj simulacija nakon nakon i-tog poteza
- c je parametar iskorištavanja, teoretski je uvijek $\sqrt{2}$
- t označava ukupan broj simulacija, koji odgovara sumi $n_i$-ova

Prvi dio formule odgovara iskorištavanju djeteta koje je do sada donijelo najviše pobjeda i poteza koji slijede nakon njega, a drugi dio proširivanju poteza koji su korišteni u malom broju simulacija. Dokazano je da procjena najboljih poteza u MCTS algoritmu konvergira minimax procjeni. Glavne prednosti ovog algoritma su što je dovoljno implementirati pravila igranja: odabir mogućih poteza iz određenog stanja i krajnje stanje igre da bi ga primijenili te što stablo pretraživanja raste asimetrično postižući tako bolje rezultate od klasičnih algoritama s velikim faktorom grananja. Algoritam se može poboljšati uvođenjem heuristika koje utječu na odabir idućeg poteza te ga je moguće paralelizirati.

**Implementacija Monte Carlo stabla pretraživanja**

Algoritam je implementiran kao funkcija UCT koja koristi strukturu stabla. Stablo se konstruira pomoću instanci klase Node. Svaka instanca klase čuva jedno stanje igre.

Funkcije članice klase Node:

- konstruktor: stvara jedan čvor stabla, inicijalizira varijable vrijednostima koje odgovaraju trenutnom stanju igre;
- UCTSelectChild(): pomoću UCT formule odabire najpogodnije dijete trenutnog čvora;
- AddChild(): dodaje dijete trenutnom čvoru;
- Update(): poziva se na kraju svake simulacije za ažuriranje znanja;

Ostale pomoćne funkcije su: ChildrenToString(), IndentString(), TreeToString(), repr() koje služe pri ispisu stabla.

Funkcija UCT je klasičnoga oblika koji se sastoji od 4 dijela navedena u opisu MCTS algoritma te koristi klasu Node za čuvanje znanja o potezima u igri.

```
[M:None W/V:26/2000 U:[]potez igra 2
| [M:0 W/V:0/13 U:[]potez igra 1
| | [M:1 W/V:6/6 U:[]potez igra 2
| | | [M:0 W/V:0/3 U:[]potez igra 2
| | | | [M:0 W/V:0/2 U:[]potez igra 1
| | | | | [M:0 W/V:1/1 U:[]potez igra 2
| | | [M:1 W/V:0/2 U:[]potez igra 2
| | | | [M:0 W/V:0/1 U:[0]potez igra 1
| | [M:0 W/V:6/6 U:[]potez igra 2
| | | [M:1 W/V:0/3 U:[]potez igra 2
| | | | [M:0 W/V:0/2 U:[]potez igra 1
| | | | | [M:0 W/V:1/1 U:[]potez igra 2
| | | [M:0 W/V:0/2 U:[]potez igra 1
| | | | [M:0 W/V:1/1 U:[0]potez igra 2
| [M:2 W/V:1974/1974 U:[]potez igra 2
| | [M:0 W/V:987/987 U:[]potez igra 1
| | | [M:0 W/V:0/493 U:[]potez igra 2
| | | | [M:0 W/V:492/492 U:[]potez igra 1
| | | | | [M:0 W/V:0/491 U:[]potez igra 2
| | | [M:1 W/V:0/493 U:[]potez igra 2
| | | | [M:0 W/V:492/492 U:[]potez igra 1
| | | | | [M:0 W/V:0/491 U:[]potez igra 2
| | [M:1 W/V:986/986 U:[]potez igra 1
| | | [M:1 W/V:0/493 U:[]potez igra 2
| | | | [M:0 W/V:492/492 U:[]potez igra 1
| | | | | [M:0 W/V:0/491 U:[]potez igra 2
| | | [M:0 W/V:0/492 U:[]potez igra 2
| | | | [M:0 W/V:491/491 U:[]potez igra 1
| | | | | [M:0 W/V:0/490 U:[]potez igra 2
| [M:1 W/V:0/13 U:[]potez igra 1
| | [M:1 W/V:6/6 U:[]potez igra 2
| | | [M:1 W/V:0/3 U:[]potez igra 2
| | | | [M:0 W/V:0/2 U:[]potez igra 1
| | | | | [M:0 W/V:1/1 U:[]potez igra 2
| | | [M:0 W/V:0/2 U:[]potez igra 1
| | | | [M:0 W/V:1/1 U:[0]potez igra 2
| | [M:0 W/V:6/6 U:[]potez igra 2
| | | [M:0 W/V:0/3 U:[]potez igra 1
| | | | [M:0 W/V:2/2 U:[]potez igra 2
| | | | | [M:0 W/V:0/1 U:[]potez igra 2
| | | [M:1 W/V:0/2 U:[]potez igra 2
| | | | [M:0 W/V:0/1 U:[0]potez igra 1
```

*Illustration 2: UCT stablo pretraživanja 3 koraka prije kraja*

## 3. Implementacija igre

Igru smo odlučili implementirati po uzoru na objektno programiranje, tako da se ona brine o svim elementima vezanim uz samu igru. Tako ona sadrži pravila potrebna za igranje, te sve ostale funkcije potrebne za ažuriranje trenutnog stanja. Tako klasa briškula ima elemente u kojima ima pohranjene bodove svih igrača te sve karte, te zna koje su karte izašle te koje su još uvijek u igri. Također se brine o podjeli karata i grafičkom sučelju. Slijedi kratak opis korištenih funkcija.

Funkcije članice klase briškula

- konstruktor: postavlja varijable klase na početne vrijednosti, klasa se poziva s parametrom koji označava broj igrača, trenutna implementacija podržava samo igranje 2 igrača, ali u planu je nadogradnja igrice za mogućnost i za 4 igrača;
- pocetak(): konstruktor grafičkog sučelja;
- play(): ispisuje na ekran početni prozor na kojem je moguće pokrenuti igru;
- postavi_karte_na_stol(): iscrtava karte igrača te špil;
- ekran(): iscrtava bačene karte;
- igra_comp(): pomoću UCT algoritma vrši odabir karte za bacanje kod računala;
- igra_covjek(): čeka na potez igrača koji se vrši klikom na jednu od karata;
- kraj_runde(): postavlja završni ekran igre, daje igraču mogućnost odabira nove igre ili zatvaranje prozora;
- Clone(): radi duboku kopiju stanja klase;
- DoMove: brine se o trenutnom stanju igre; svako bacanje karte poziva DoMove koji koristeći pomoćne funkcije određuje je li runda gotova te u tom slučaju određuje pobjednika ( samim time igrača koji je na potezu ), ažurira bodove te ako mora podjeli karte.

Funkcije koje DoMove poziva su: baci_kartu, update_pobjednika_i_bodova, podijeli_karte i podjeli karte_zadnji_put;

- GetMoves(): određuje koje sve poteze UCT može odigrati, možda se može poboljšati tako da se određeni potezi zabrane te tako smanji stablo pretraživanja
- GetResult(): određuje pobjednika igre;
- trenutno_uzima(): određuje koji igrač ima najjaču kartu na stolu i broj bodova koje uzima
- dodjeli_kartu(): dodjeljuje random kartu;

Ostale pomoćne funkcije su: koja_je_to_karta(), tko_je_pobijedio(), je_li_briskula(), poeni(), postavi briskulu(), podijeli_karte_na_pocetku(), print1().

## 4. Testiranje algoritma

Algoritam smo osim igranjem protiv njega, radi lakšeg i bržeg testiranja odlučili testirati na 3 načina:

- Borbom protiv samog sebe s različitim brojem iteracija
- Borbom protiv heuristike koja simulira igranje čovjeka
- Borbom protiv samog sebe s različitim koeficijentom c u UCT formuli

1. U borbi protiv samog sebe pokušali smo pronaći optimalan broj iteracija tako da se na odgovor algoritma ne čeka previše, ali opet da bude dobro obaviješten. Nakon testiranja na 50 igara dobili smo slijedeće rezultate:

| Broj iteracija UCT0 | Broj iteracija UCT1 | Broj pobjeda UCT1 |
|---|---|---|
| 300 | 500 | 25 |
| 500 | 750 | 24 |
| 300 | 1000 | 35 |
| 500 | 1000 | 29 |
| 800 | 1000 | 33 |
| 900 | 1000 | 23 |
| 950 | 1000 | 23 |
| 1000 | 1500 | 25 |
| 300 | 2000 | 33 |
| 500 | 2000 | 28 |
| 750 | 2000 | 29 |
| 1000 | 2000 | 29 |
| 1250 | 2000 | 33 |
| 1500 | 2000 | 27 |
| 1750 | 2000 | 30 |
| 2000 | 2500 | 27 |
| 2000 | 3000 | 24 |
| 2000 | 4000 | 21 |
| 500 | 5000 | 22 |
| 2000 | 5000 | 19 |
| 2000 | 10000 | 26 |

Iz tablice se vidi da je optimalan broj iteracija oko 2000. Ako mu suprotstavimo UCT s manjim brojem iteracija primjećujemo da UCT s 2000 iteracija uvijek pobijedi više od 50% puta, dok u suprotnom slučaju kad se bori protiv UCT-a s više iteracija rezultat nije predvidljiv.

2. U borbi s heuristikom u 50 igara dobili smo iduće rezultate:

| Broj iteracija UCT | Broj pobjeda UCT |
|---|---|
| 100 | 38 |
| 150 | 42 |
| 200 | 39 |
| 200 | 43 |
| 250 | 42 |
| 300 | 44 |
| 400 | 45 |
| 600 | 44 |
| 700 | 46 |
| 800 | 46 |
| 900 | 45 |
| 1000 | 47 |
| 2000 | 44 |
| 5000 | 44 |

Iz tablice je moguće očitati da borba UCT-a i heuristike (detaljnije opisana kasnije) završi gotovo uvijek jednako, to jest UCT uvijek prevlada, ali bez obzira na broj iteracija, kako briškula ipak ima udio sreće nikad ne uspije pobijediti u 100% slučajeva.

Heuristiku koju smo koristili željeli smo učiniti da bude što bliža našem načinu igranja.
Osim toga kako znamo koje su karte izašle, možemo i lagano računati vjerojatnosti koje nam nisu baš moguće kod stvarnog igranja.
Kod nje razlikujemo dvije situacije, kad je heuristika prva, odnosno druga na redu.

- Kad je heuristika prva na redu prvo izračuna za svaku kartu koju u ruci kolika je vjerojatnosti da protivnik ima jaču od nje.
  Ako je u posjedu karte koja nije briškula, i ne postoji jače od nje, nju bacamo (ako ih ima više bacamo onu s najviše bodova).
  Ako nemam takvu kartu onda baca kartu s najviše bodova (ne kariga i briškulu) za koju ne postoji karig iste boje u igri.

Ako nemamo ni takve karte onda baca proizvoljan lišo, u slučaju da nema lišo baca bodove, i to sto vise u slučaju da su oba kariga izašla.
U slučaju da nema ni takvu kartu onda će pokušati baciti lišo briškulu.
Ako ne postoji ni takva karta onda se odlučuje na bacanje kariga (ako je vjerojatnst da ima briškulu manja od 50%).
Inače baca briškulu kariga, ili ako ga nema onda baca proizvoljan karig

- Kad je heuristika druga na redu za bacanje karte, prvo provjeri je li moguće uzeti bez da baci briškulu. Ako je uzima i to s najviše bodova što može.
  Ako je u nemogućnosti uzeti provjerava odlučuje se za dvije stvari: želi li uzeti ili ne.
  Heuristika želi uzeti ukoliko je broj bodova na stolu veći od 4, ili ako je drugi igrač s bodovima na stolu i onime sto će mu morat dati pobjednik.
  Ako je slučaj takav da želi uzeti i ne može onda daje najmanje moguće bodova.
  U slučaju da heuristika želi pustiti, a to je ako će drugi igrač uzeti u trenutnoj rundi manje od 6 bodova, heuristika popušta s najmanjim mogućim gubitkom.

3. Borbom UCT-a s istim brojem iteracija, a različitim koeficijentima pri čemu UCT0 ima fiksiran koeficijent $\sqrt{2}$ dobijemo sljedeće podatke:

| Koeficijent u UCT formuli | Broj pobjeda / Broj igara |
|---|---|
| 0.01 | 6/50 |
| 0.1 | 12/50 |
| 0.4 | 26/50 |
| 0.6 | 26/50, 27/50, 51/100, 102/200, 94/200 |
| 0.8 | 27/50, 25/50, 27/50, 107/200, 89/200 |
| 1 | 23/50 |
| 1.4 | 21/50 |
| 2 | 21/50 |
| 10 | 22/50 |
| 20 | 18/50, 23/50, 27/50 |
| 200 | 24/50 |
| 1000 | 18/50 |

Iz dobivenih podataka zaključili smo da veličina izabranoga koeficijenta c ukoliko je on u intervalu između 0.4 i 20 ne mijenja bitno situaciju, te zbog nausumičnosti podjele karata broj pobjeda varira.

## 5. Zaključak

Iz različitih testiranja MCTS algoritma dolazimo do zaključka da je algoritam jako pouzdan za ovu igru. Prednost je što ne zahtjeva nikakve složene heurstike i daje dobre rezultate već pri niskom broju iteracija. Najpouzdanije testiranje, kada čovjek igra protiv računala, dalo je dobre rezultate. U većini slučajeva čovjek izgubi. Algoritam bi se mogao poboljšati tako da osim nastojanja u tome da pobjedi pokuša i maksimizirati konačan broj bodova. Osim toga, mogla bi se uvesti heuristika koja će mu ograničiti broj poteza mogućih u slijedećem potezu. U slučaju pametne heuristike, izbjeglo bi se preveliko grananje stabla, i ubrzalo računanje.

## 6. Link na igricu:

Za pokrenuti igru treba: skinuti i raspakirati mapu sa sljedećeg linka:
https://www.dropbox.com/sh/ov1h0n80qz5bjs1/AADW-WXMS19bHOGkx4DkhDYWa?dl=0
Na izboru su 4 opcije igranja: "Briskula_easy.exe" ( 200 iteracija ), "Briskula_medium.exe" ( 2000 iteracija ) i "Briskula_hard.exe" (5000 iteracija) te "Briskula_otovorene_karte.exe" ( 5000 iteracija ).
Za pokretati igricu u pythonu potrebno je osim Pythona 2.7 instalirati iz mape "dodatne instalacije" "pygame-1.9.2a0.win32-py2.7" za grafičko sučelje, te pokrenuti "igra_zatvorene_karte.py" ili "igra_otvorene_karte.py".
Za igru UCT-a protiv heuristike potrebno je pokrenuti "simulacija.py".
Za igru UCT protiv UCT-a treba pokrenuti "UCT_vs_UCT.py".
Za izradu .exe datoteka iz pythona treba prvo instalirati iz mape "dodatne instalacije" "py2exe-0.6.9.win32-py2.7", te iz komandne linije pokrenuti naredbu "pygame setup.py py2exe".

## 7. Literatura:


https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

http://mcts.ai/index.html

http://pygame.org/hifi.html

https://pygame.org/wiki/tutorials