

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

EKSPERTNI SUSTAVI
PROJEKT

Ekspertni sustav za igru “Connect Four” u programskom jeziku LISP

Mateja Čuljak

Domagoj Bulat

Zagreb, siječanj 2012.

1 Uvod

U sklopu ovog projekta razvijen je ekspertni sustav za igranje igre Connect Four. Connect Four je igra za dva igrača na ploči dimenzija 6×7 . Svaki igrač ima kuglice jedinstvene boje (ili npr. znakove jedinstvene za tog igrača), te igrači naizmjenice ubacuju kuglicu u ploču. Pritom se kuglice mogu ubaciti samo na vrh nekog od stupaca te padaju na prvo slobodno mjesto odozdo, odnosno možemo reći da se igra uz utjecaj gravitacije. Budući da se kuglice ubacuju na vrh stupaca, svaki put kad je na redu igrač može odabrati do 7 različitih poteza (manje od 7 ukoliko su neki stupci već popunjeni). Cilj igre je skupiti 4 vlastite kuglice u nizu u bilo kojem smjeru, uključujući i dijagonalno.

2 Opis rješenja

Ekspertni sustav razvijen unutar ovog projekta se može okarakterizirati kao inteligentni agent baziran na cilju te korisnosti.

Sustav koristi pretraživanje prostora stanja i planiranje, odnosno pokušava odrediti posljedice vlastitih akcija, te koristi računanje funkcije korisnosti (ili dobrote) kako bi odredio moguću kvalitetu pojedinih akcija.

Računalo za svaki potez može odabrati od 1 do 7 stupaca (ovisno o popunjenosti ploče). Radi odabira poteza vrši se *pretraživanje u dubinu*. Ako se promatra najsloženiji slučaj, početno stanje ploče, na prvoj razini imamo 7 mogućnosti, na drugoj dodatnih 7 za svaku mogućnost s prve razine, itd. Znači, na n -toj razini imamo 7^n mogućnosti. Budući da broj mogućnosti raste eksponencijalno potrebno je ograničiti pretraživanje s obzirom na dubinu. Pokazalo se da se pretraživanje do dubine 5 izvodi dovoljno brzo te dovodi do kvalitetnih zaključaka.

Kvaliteta poteza se određuje po minimax algoritmu. Cilj algoritma je maksimizirati dobitak te minimizirati gubitak. Algoritam pretpostavlja da drugi igrač igra optimalno. Pri pretraživanju stanja ploče pamtimo tko u trenutnom stanju odabire stupac. Ako se radi o stanju u kojem računalo igra (pokretač algoritma), odabiremo potez koji ima najveću vrijednost. Ako smo u stanju u kojem protivnik (čovjek) igra, odabiremo sljedeće stanje s minimalnom vrijednošću jer znamo da bi protivnik to odabrao (jer igra optimalno). Algoritam se može opisati sljedećim pseudokodom:

```
MINIMAX(ploca, igrac, dubina):
    IF dubina = 5:
        return heuristika(ploca)
    ELSE:
        IF pobjeda(ploca, HUMAN):
            return -1
        IF pobjeda(ploca, COMPUTER):
            return 1
        IF igrac = COMPUTER:
            return MAX(MINIMAX(novaPloca(ploca, 1), HUMAN, dubina+1),
                       ..., MINIMAX(novaPloca(ploca, 7), HUMAN, dubina+1))
        ELSE:
            return MIN(MINIMAX(novaPloca(ploca, 1), COMPUTER, dubina+1),
                       ..., MINIMAX(novaPloca(ploca, 7), COMPUTER, dubina+1))
```

pri čemu funkcija `novaPloca(ploca, n)` vraća novu ploču koja je dobivena ubacivanjem žetona u n -ti stupac ploče `ploca`, funkcija `heuristika(ploca)` vraća procjenu kvalitete ploče, a funkcija `pobjeda(ploca, igrac)` vraća istinitosnu vrijednost koja govori je li igrač ostvario pobjedu na ploči. U slučaju pobjede na određenoj dubini, funkcija `MINIMAX` vraća 1, u slučaju poraza -1. U rješenju je heuristika postavljena na konstantu (0), jer se pokazala nepotrebnom (algoritam je dovoljno dobar i bez heuristike). U slučaju da se na izlazu minimaxa pojavi više jednakih ekstrema, odabire se prvi. Razne implementacije minimaxa u tom slučaju odabiru nasumični potez, ali tu dolazimo do problema izračuna pseudo slučajnih brojeva u funkcijskim jezicima, takav izračun zahtjeva održavanje stanja (jezgre generatora, *seeda*), a to funkcijski jezici žele zaobići (no nije neizvedivo).

Zbog specifičnosti igre, odnosno situacijama kad su oba igrača blizu kraja, bilo je potrebno modifirati algoritam te provesti dvije provjere prije izračuna same kvalitete rješenja. Radi se o provjerama

postoji li potez koji sprječava poraz, te provjeri postoji li potez koji vodi direktno u pobjedu. Provjera pobjede se mora napraviti jer bez nje algoritam postaje previše defenzivan tj. glavni cilj mu postaje sprječavanje protivnika u pobjedi umjesto osiguravanja vlastite pobjede, te se može dogoditi da propusti sigurnu pobjedu.

3 Opis implementacije

Implementacija je izvedena koristeći programski jezik Clojure (<http://www.clojure.org/>), dijalekt LISP-a. Clojure se izvodi na Java virtualnom stroju (engl. *Java Virtual Machine*, JVM) te omogućava korištenje Javinih biblioteka. Ta karakteristika je iskorištena pri izradi korisničkog sučelja (GUI) za što je korišten *Swing* (skup Java biblioteka za izradu grafičkog sučelja). Najbitnije funkcije u implementaciji su:

gui: ulazna točka programa, pokretanje GUI-a,

Ploca: definicija ploče izvedena pomoću asocijativnog niza, dotiče se problema održavanja promjenjivog stanja u funkcijskim jezicima, stoga je korišten konstrukt Clojura `atom` koji omogućava transparentno održavanje stanja,

odigraj: odigravanje poteza na ploči,

provjeriSvaki4: provjera pobjede na 4 polja u svakom smjeru počevši od zadanog,

najbolji-potez: traženje najkvalitetnijeg poteza na ploči,

kvaliteta-poteza: određivanje kvalitete poteza do određene dubine.

Jedan od bitnih parametara je dubina stabla. Dubina je definirana na početku programa naredbom: (`def dubina 5`). Veća dubina nam osigurava dublju pretragu stabla, tj. bolju igru ekspertnog sustava. U slučaju dubine vrijednosti 5 sustav pretražuje 5 budućih poteza. Iako se dubina vrijednosti 5 pokazala kao dovoljno dobra, dubinu je moguće povećati, ili smanjiti ukoliko se potezi računaju presporo (ovisno o računalu na kojem se program izvršava).

4 Upute za korištenje

Prije pokretanja programa potrebno je instalirati Clojure. Za instaliranje Clojura potrebno je imati instaliran Java Runtime Environment, koji je moguće skinuti na www.oracle.com/technetwork/java/javase/downloads/index.html. Clojure je moguće skinuti na *web* sjedištu www.clojure.org/downloads. Skinuti paket u kojem se nalazi Clojure je potrebno otpakirati, te u tom direktoriju pokrenuti naredbu (unutar komandne linije ili terminala):

```
java -cp clojure-1.3.0.jar clojure.main
```

Datoteka `clojure-1.3.0.jar` može imati nešto drukčiji naziv ovisno o verziji Clojura koji se nalazi na računalu. Time smo ušli u REPL (engl. *read-eval-print loop*) te možemo isprobati rad Clojura.

Implementirani ekspertni sustav se nalazi u datoteci `ploca.clj`. Sustav možemo pokrenuti naredbom

```
java -jar clojure-1.3.0.jar <putanja_do_ploca.clj>
```

Pokretanjem te naredbe ulazimo u grafičko sučelje sustava, te možemo započeti igranje igre protiv ekspertnog sustava. Klikom na gumbove “Ubaci” moguće je ubaciti svoj znak u određeni stupac. Grafičko sučelje je prikazano na slici 1. U slučaju pobjede jednog od igrača, pojavit će se novi prozor s porukom “Pobijedio je: X” (ili “Pobijedio je: O”).

5 Zaključak

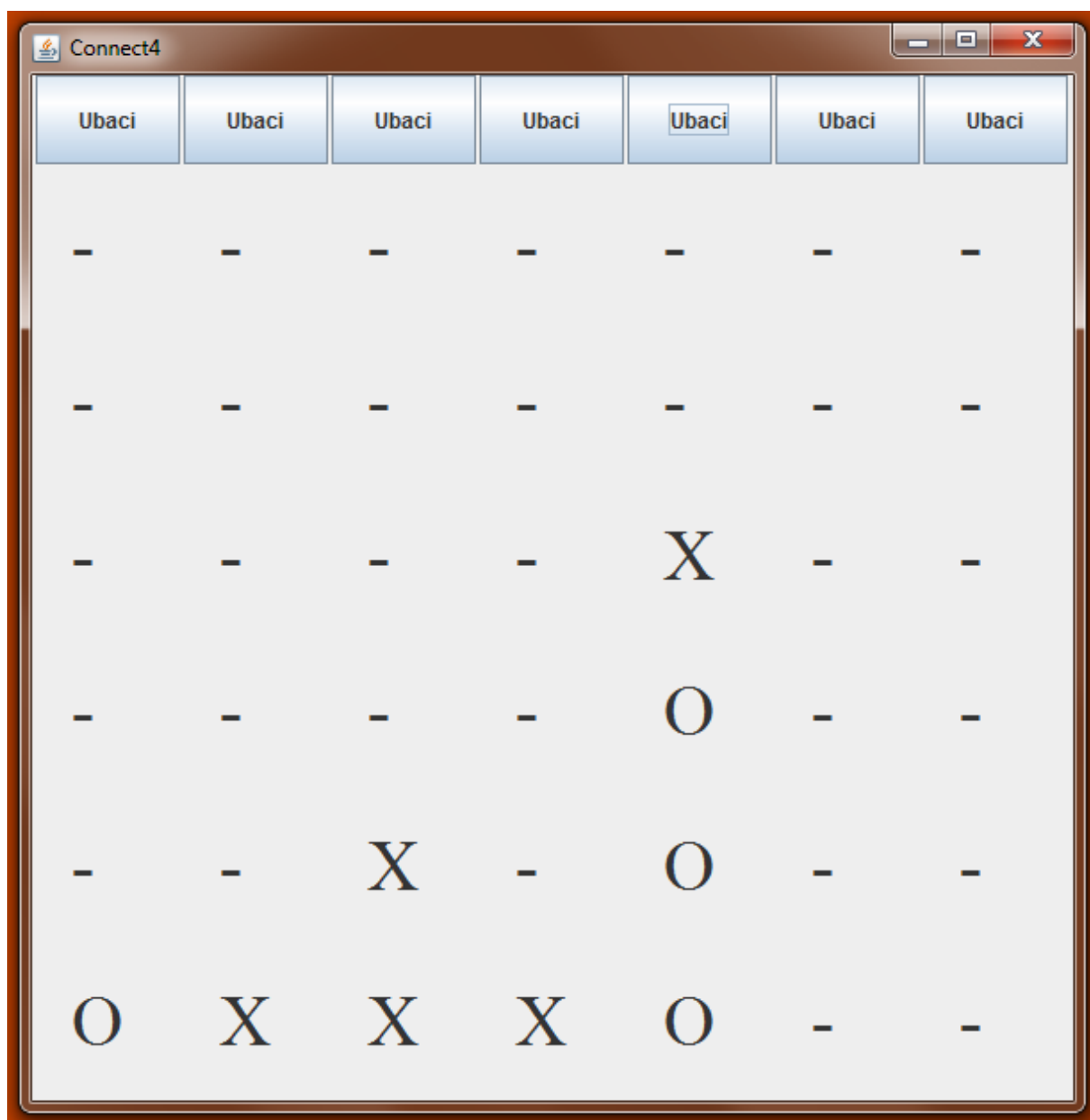
Ovaj projekt je pokazao uporabu programskog jezika LISP u korištenju ekspertnih sustava, posebno u području igranja igara baziranih na pretraživanju prostora stanja i računanju kvalitete stanja. Korišteni dijalekt LISP-a, Clojure, se pokazuje kao koristan zbog izvođenja na Java virtualnom stroju što omogućava lako kombiniranje s velikim brojem postojećih biblioteka programskog jezika Java.

Nedostatak programskog jezika LISP u ovom području je problematičan rad s održavanjem stanja u usporedbi s ostalim programskim paradigmatama. S druge strane, LISP se pokazao kao odličan izbor za problem pretraživanja stanja. Zbog svoje referencijalne prozirnosti i izbjegavanja popratnih efekata¹ (engl. *side effects*), pri stvaranju novog stanja ne moramo razmišljati o njegovom utjecaju na ostala stanja. Ako promotrimo pretraživanje stanja na nekoj razini, možemo vidjeti da generiramo 7 novih ploča. U imperativnom jeziku poput C-a, za potrebu generiranja novog stanja mi bismo:

1. promijenili trenutno stanje ($s \rightarrow s'$), izvršili rekurzivni poziv, uklonili promjenu stanja ($s' \rightarrow s$) radi sljedećeg rekurzivnog poziva (radimo u istom memorijskom prostoru); ili
2. kopirali stanje s u novi memorijski prostor ($s' := s$), promijenili stanje s' ($s' \rightarrow s''$), izvršili rekurzivni poziv, oslobodili memoriju stanja s'' (radimo u različitom memorijskom prostoru).

Radi referencijalne prozirnosti koju nam funkcijski jezici pružaju navedene “kućanske poslove” možemo zanemariti.

¹Radi se o kršenju idempotentnosti. Primjer problema koji uzrokuju popratni efekti, promotrimo kod: “`x = 1; void y() { x++; return x; };`” Matematički gledano, poziv funkcije uz jednake parametre mora dati jednak rezultat. U navedenom slučaju izraz “`y() == y()`” vraća logičku neistinu!



Slika 1: Prikaz grafičkog sučelja ekspertnog sustava.