SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2858

RAZVOJ AGENTA ZA DRUŠTVENU IGRU CATAN

Ivan Skorić

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2858

RAZVOJ AGENTA ZA DRUŠTVENU IGRU CATAN

Ivan Skorić

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 2858

Pristupnik:	**Ivan Skorić (0036508459)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Razvoj agenta za društvenu igru Catan**

Opis zadatka:

Proučiti pravila društvene igre Catan. Istražiti postojeće programske okvire koji omogućuju igranje zadane igre kao i mogućnost proširenja s novim strategijama za računalne agente. Proučiti metode koje su u literaturi bile korištene za razvoj novih agenata za igru Catan. Proučiti postojeće agente u razvijenom okviru te ustanoviti moguća područja u kojima se oni mogu poboljšati. Predložiti strategije za pojedine dijelove igre. Razviti nove agente za igru Catan na temelju predloženih strategija te ocijeniti njihovu uspješnost u usporedbi s postojećim agentima kao i s ljudskim igračima. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 27. lipnja 2022.

Zahvaljujem mentoru doc. dr. sc. Marku Đuraseviću na uputama, savjetima i kvalitetnoj suradnji tijekom izrade programskog rješenja te pisanja diplomskog rada. Također zahvaljujem obitelji, prijateljima i kolegama koji su mi pružali podršku i razumijevanje tijekom čitavog studija.

Sadržaj

Uvod

Umjetna inteligencija (engl. *artificial intelligence*, AI) trenutno je jedno od najbrže rastućih domena računarstva. U suštini, pojam umjetne inteligencije može se definirati kao reproduciranje prirodne inteligencije ljudi i životinja u strojevima ili računalnim sustavima. Glavnim karakteristikama inteligencije smatraju se sposobnost učenja, donošenja odluka i zaključaka, te rješavanje problema. Umjetna inteligencija čini temelje brojnih novih dostignuća u razvoju područja poput robotike, autonomne vožnje, prepoznavanja govora, analize financijskih sustava, medicinske dijagnoze, upravljanja IoT (Internet stvari, engl. *Internet of Things*) sustava, te mnogih drugih. Navedene tehnologije predstavljaju velike dijelove naših života, no jedno od zanimljivijih područja kroz koje možemo pratiti napredak umjetne inteligencije je igranje igara. Društvene i videoigre imaju jasno definirana pravila i ciljeve, te zato čine odlično okruženje za razvoj i analizu AI tehnologije. Od IBM-ovog šahovskog genija Deep Blue, preko virtuoza društvenih igri AlphaZero, sve do velemajstora videoigre StarCraft II AlphaStar, mnogi popularni uspjesi u savladavanju strateških igara predstavljaju najbolje tehnike iz domene umjetne inteligencije za svoje vrijeme.

Cilj ovog rada je primijeniti metode iz područja umjetne inteligencije i strojnog učenja u razvoju nekoliko različitih računalnih igrača, odnosno agenata za stratešku društvenu igru Catan, te analizirati njihov uspjeh i ponašanje u igri. Prvo poglavlje čini upoznavanje s pravilima i konceptima igre. U drugom poglavlju, izneseni su zahtjevi, te je istraženo nekoliko radnih okvira za igranje Catana na računalu, i za daljnji razvoj. Treće poglavlje detaljno opisuje programsko rješenje u sklopu ovog rada. Definirani su i obrazloženi načini donošenja odluke kod pojedinih agenata, te su opisane metode strojnog učenja korištene prilikom usavršavanja istih. Također, navedene su dorade napravljene nad implementacijom odabranog radnog okvira. Četvrto poglavlje bavi se usporedbom rezultata koje postižu implementirani agenti, te iznosi zapažanja o sposobnostima agenata prilikom testiranja. U petom poglavlju, predloženo je nekoliko ideja za budući razvoj i napredak ovog projekta, te su navedene izmjene za kojima se ukazala potreba prilikom razvoja.

1. Društvena igra Catan

Catan[1], prije poznat kao The Settlers of Catan, je strateška društvena igra izdana 1995. godine u Njemačkoj prema dizajnu Klausa Teubera. Do sada (2022.) prodano je više od 40 milijuna primjeraka igre, i procjenjuje se da igra ima oko 20 milijuna redovnih igrača [1]. Sljedeći tekst je kratak uvod u opis igre sa službenih web-stranica Catana.

> *Picture yourself in the era of discoveries: after a long voyage of great deprivation, your ships have finally reached the coast of an uncharted island. Its name shall be Catan! But you are not the only discoverer. Other fearless seafarers have also landed on the shores of Catan - the race to settle the island has begun!* [2]

Igra se odvija na otoku Catanu, a cilj je ostvariti najveći napredak izgradnjom naselja, gradova, cesta i vojske uz vješto upravljanje resursima. Igru obilježava sustav razmjene resursa – igrači mogu proizvoljno dogovarati međusobnu trgovinu resursa, ili se osloniti na unaprijed definirane razmjene putem pomorske trgovine, bez sudjelovanja drugih igrača. Promišljenim smještajem naselja i gradova, te lukavom trgovinom, igrač može ostvariti veliku prednost u svom razvoju.

1.1. Pravila igre

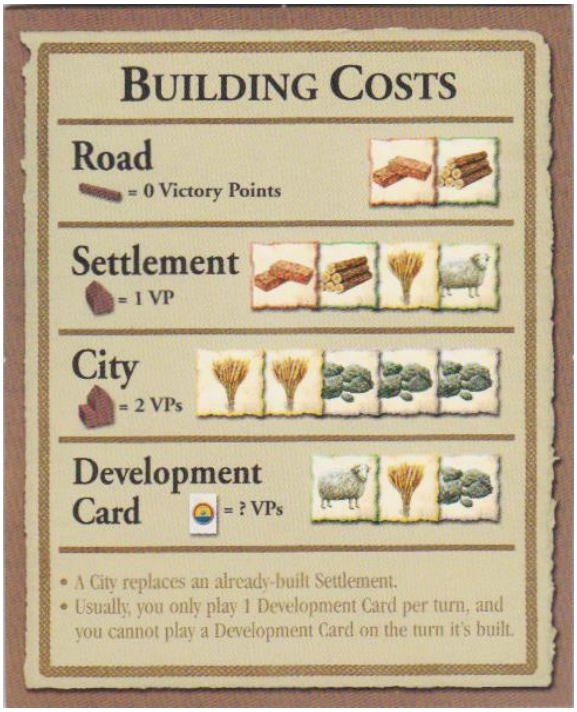
U osnovnom izdanju, igra je preporučena za 3 do 4 igrača. Otok Catan, kojeg igrači naseljavaju, sačinjen je od 19 šesterokutnih pločica raznovrsnog terena i okružen je morem. Vrste terena su sljedeće: 4 šume, 4 polja, 4 pašnjaka, 3 brda, 3 planine i 1 pustinja. Svaka vrsta terena osim pustinje generira određen resurs – šume generiraju drvo, polja žito, pašnjaci ovce odnosno vunu, brda cigle, a planine kamenu rudu. Na pločice terena postavljaju se oznake s brojevima od 2 do 12, tj. mogućim ishodima bacanja dvije standardne kockice (vidi Slika 1.1).

[1] https://www.catan.com/

Slika 1.1 *Primjer postavljene igre Catan* [2]

Svaki potez započinje bacanjem kockica, povodom kojeg pločice terena označene dobivenim brojem proizvode odgovarajući resurs. Igrači mogu prisvojiti proizvodnju resursa gradnjom naselja na vrhovima između pločica terena. Time će, u fazi proizvodnje, za svako naselje dobiti 1 karticu resursa proizvedenog na susjednoj pločici. Nadogradnjom naselja u grad, počinju dobivati 2 kartice proizvedenog resursa. Najvjerojatniji ishod bacanja kockica – broj 7 – ne proizvodi resurse, već igra posebnu ulogu. Povodom bacanja broja 7, svi igrači s više od 7 kartica resursa u svojoj ruci moraju odabrati pola kartica (zaokruženo na manje) i vratiti ih u „banku“. Igrač koji je na svom potezu bacio 7 mora pomaknuti figuru razbojnika na novu pločicu. Razbojnik započinje igru na pločici pustinje, a njegovo prisustvo na bilo kojoj drugoj pločici sprječava proizvodnju resursa na istoj, dokle god se tamo nalazi. Dodatno, igrač koji ga je pomaknuo smije ukrasti 1 nasumičan resurs bilo kojem igraču s naseljem ili gradom na odabranoj pločici. Nakon faze proizvodnje, igrač može koristiti svoj potez za izgradnju i trgovinu.

Igrač troši svoje resurse na izgradnju naselja, gradova i cesta, te na kupovinu razvojnih kartica (engl. *development cards*) (vidi Slika 1.2). Broj figura za gradnju ograničen je na 5 naselja, 4 grada i 15 cesta po igraču. Igra započinje s dva kruga inicijalne izgradnje naselja i cesta. U tim krugovima, svaki igrač gradi jedno naselje na proizvoljnom vrhu, odnosno križanju između pločica terena, uz uvjet da su svi susjedni vrhovi prazni. Neposredno do naselja, gradi cestu na jednom od slobodnih bridova. U drugom krugu inicijalne izgradnje, igrači igraju obrnutim redoslijedom (posljednji igrač gradi prvi), te dobivaju po jedan resurs sa svake susjedne pločice drugog naselja. Kod svake buduće gradnje, cesta se smije graditi samo na slobodnim bridovima koji se nastavljaju na postojeću vlastitu cestu, naselje ili grad. Naselje se smije graditi samo na slobodnom vrhu do kojeg već postoji igračeva cesta, uz isti uvjet da su svi susjedni vrhovi prazni. Grad se može graditi isključivo na mjestu postojećeg naselja, odnosno naselje se može nadograditi u grad. Set od 25 razvojnih kartica koje igrači mogu kupovati čini 14 vitezova, 6 kartica napretka (engl. *progress cards*) i 5 kartica pobjedničkih bodova (engl. *victory points*). Igrač može odigrati karticu viteza ili napretka u bilo kojem svom potezu nakon poteza kad je kupljena, no samo jednu po potezu. Kartica viteza aktivira ranije opisanu funkcionalnost pomicanja razbojnika, a kartice napretka uključuju različite pogodnosti vezane uz dobivanje resursa ili gradnju.



Slika 1.2 *Cijene gradnje* [3]

Na svom potezu, igrač može pokrenuti trgovinu s bilo kojim drugim igračem. U igri se ovo naziva „domaćom trgovinom“ (engl. *domestic trade*). Igrač predlaže razmjenu određenih resursa, a ostali igrači mogu prihvatiti odnosno odbiti ponudu, ili dati protuponudu. Osim međusobne trgovine, igrač može napraviti razmjenu s „bankom“, bez sudjelovanja drugih igrača. Ovo se naziva „pomorskom trgovinom“ (engl. *maritime trade*). Standardna razmjena dostupna svim igračima je 4 istovjetna resursa za 1 drugi resurs. Ako igrač ima naselje ili grad s pristupom luci, ima mogućnost povoljnije razmjene s „bankom“. Na obalama Catana nalazi se 9 luka i svaka je vezana uz dva susjedna vrha s kojih joj igrači mogu pristupiti. 5 luka je specifično za pojedinu vrstu resursa, te nude razmjenu 2 kartice tog resursa za 1 drugi resurs. Preostale 4 luke su univerzalne i omogućuju razmjenu 3 istovjetna resursa za 1 drugi resurs.

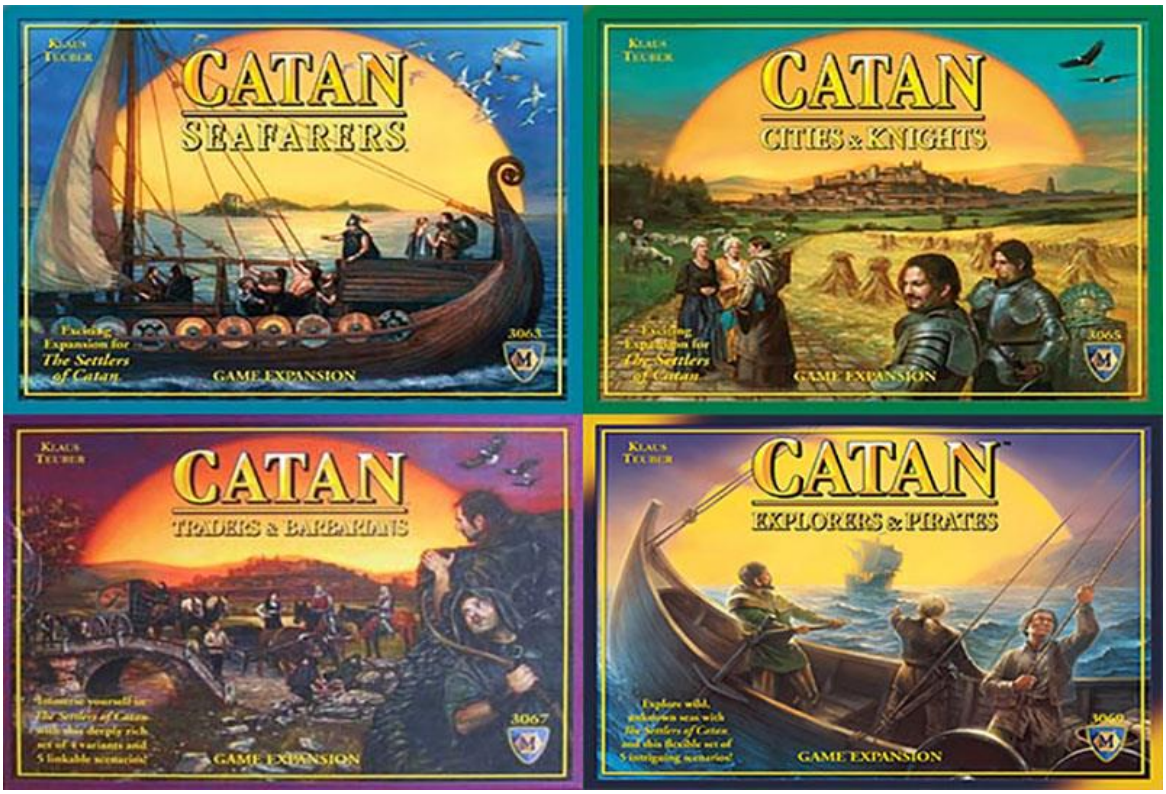
Pobjednik igre je igrač koji prvi sakupi 10 pobjedničkih bodova. Svaki grad igraču nosi 2 boda, a svako naselje 1. Postizanjem dužine od 5 bridova, igrač s najdužom cestom dobiva 2 dodatna boda, dokle god posjeduje najdužu cestu. Za dužinu ceste uzima se u obzir samo jedan najduži neprekinuti put, bez prolaska kroz protivnička naselja ili gradove te bez pribrajanja ogranaka. Drugi igrač može preuzeti nagradu za najdužu cestu tek kad izgradi dužu (vidi Slika 1.3). Slično se nagrađuje posjedovanje najveće vojske – igrač s najviše odigranih razvojnih karata viteza nakon trećeg dobiva 2 dodatna boda. Drugi igrači također mogu preuzeti nagradu kada ga nadmaše (vidi Slika 1.4). Za kraj, igrač ne otkriva kupljene razvojne karte s pobjedničkim bodovima do kraja igre, ali pribraja ih svojim bodovima.

Slika 1.3 *Nagrada za najdužu cestu* [4]

Slika 1.4 *Nagrada za najveću vojsku* [4]

1.2. Proširenja za igru

1996. godine izdano je proširenje za Catan koje povećava područje otoka i omogućuje igru do 6 igrača. U svrhu ubrzanja igre s većim brojem igrača, uvedeno je pravilo da igrači mogu graditi ili trgovati s „bankom“ i na tuđim potezima. Kroz iduće godine, izdana su još četiri proširenja koja dodaju različite koncepte igranja, resurse, scenarije ili lokacije. To su proširenja Seafarers[2], Cities & Knights[3], Traders & Barbarians[4] i Explorers & Pirates[5] (vidi Slika 1.5). Ova proširenja nisu detaljno opisana u radu jer ih odabrani radni okvir za igru trenutno ne podržava, no neki od proučenih podržavaju.



Slika 1.5 *Službena proširenja za Catan* [5]

[2] https://www.catan.com/seafarers

[3] https://www.catan.com/cities-knights

[4] https://www.catan.com/traders-barbarians

[5] https://www.catan.com/explorers-pirates

2. Radni okviri za igru Catan

U sklopu ovog diplomskog rada proučeno je nekoliko radnih okvira za igranje društvene igre Catan na računalu, s mogućnošću implementiranja novih računalnih igrača, tj. agenata. Glavni kriterij je da radni okvir bude otvorenog koda (engl. *open-source*). Osim toga, očekuje se da postoji implementacija barem jednog funkcionalnog agenta, za analizu dostupnih funkcionalnosti, usporedbu ponašanja, i lakše razumijevanje programskog sučelja igre. Poželjno je da radni okvir omogućuje igranje kroz prikladno grafičko sučelje, sa i bez ljudskih igrača, ali i da može relativno brzo izvršiti velik broj igri bez sučelja, za potrebe prikupljanja statistike i učenja pojedinih agenata. Također, traži se da model stanja igre bude što jednostavniji, kako bi bilo moguće efikasno sagraditi stablo igre. Konačno, cilj je da programsko sučelje igrača bude što fleksibilnije za primjenu različitih metoda odlučivanja. Idealna struktura je da se za svaki potez, ili bar svaku akciju, poziva točno jedna metoda za igračevu odluku o nastavku igre, te da ta metoda ima pristup cijelom trenutnom stanju igre.

2.1. JSettlers

Prvi proučeni radni okvir je JSettlers[6], verzija 2.5. Riječ je o radnom okviru napisanom u programskom jeziku Java[7], a razvija ga Jeremy D. Monin još od 2005. godine. Projekt se i dalje redovito ažurira, te je verzija 3.x trenutno u eksperimentalnom stanju [6]. Izvorni kôd radnog okvira nalazi se u GitHub[8] repozitoriju [7], a prilikom njegove analize i testiranja za potrebe ovog rada koristilo se razvojno okruženje IntelliJ IDEA[9] tvrtke JetBrains[10].

---

[6] https://nand.net/jsettlers/

[7] https://dev.java/

[8] https://github.com/

[9] https://www.jetbrains.com/idea/

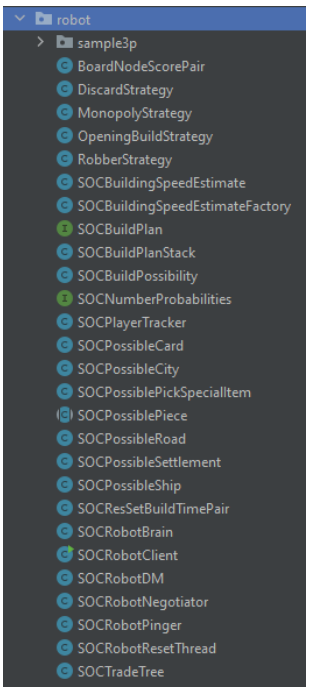
[10] https://www.jetbrains.com/

JSettlers je primarno namijenjen za igranje igara, i može se reći da u tome nadmašuje sve druge *open-source* klijente. Ima vrlo napredno korisničko sučelje kroz koje je moguće namještati brojne postavke za igru (vidi Slika 2.2), pokrenuti poslužitelj za *online* igre s drugim ljudima, spojiti se na postojeći poslužitelj, igrati *offline* igre protiv agenata, spremati stanja igre u bazu podataka, i još mnogo toga. Implementacija igre u ovom radnom okviru nudi potpunu funkcionalnost Catana. Također, implementirani su koncepti i posebni scenariji iz proširenja Seafarers, koje je moguće aktivirati u postavkama igre (vidi Slika 2.1).

U radnom okviru postoji funkcionalan agent čije se ponašanje može djelomično prilagoditi određenim parametrima. Igranjem protiv istog, može se zaključiti da je riječ o jednom od najnaprednijih agenata za Catan, no teško ga je direktno usporediti s agentima iz drugih radnih okvira koji su prvenstveno ograničeni funkcionalnostima svog klijenta. Što se tiče implementacije agenta, riječ je o vrlo kompleksnom sustavu heurističkih funkcija i nezavisnih strategija koje ne evaluiraju stvarno stablo igre, već isključivo na temelju trenutnog stanja matematički procjenjuju isplativost određenih akcija u budućnosti (vidi Slika 2.3). Složenost ovakvog pristupa izradi agenta očituje se u njegovih 12 tisuća linija kôda.

Slika 2.1 *Sučelje klijenta JSettlers u scenariju „Cloth for Catan" proširenja Seafarers* [6]

Slika 2.2 *Postavke igre u klijentu JSettlers* [6]



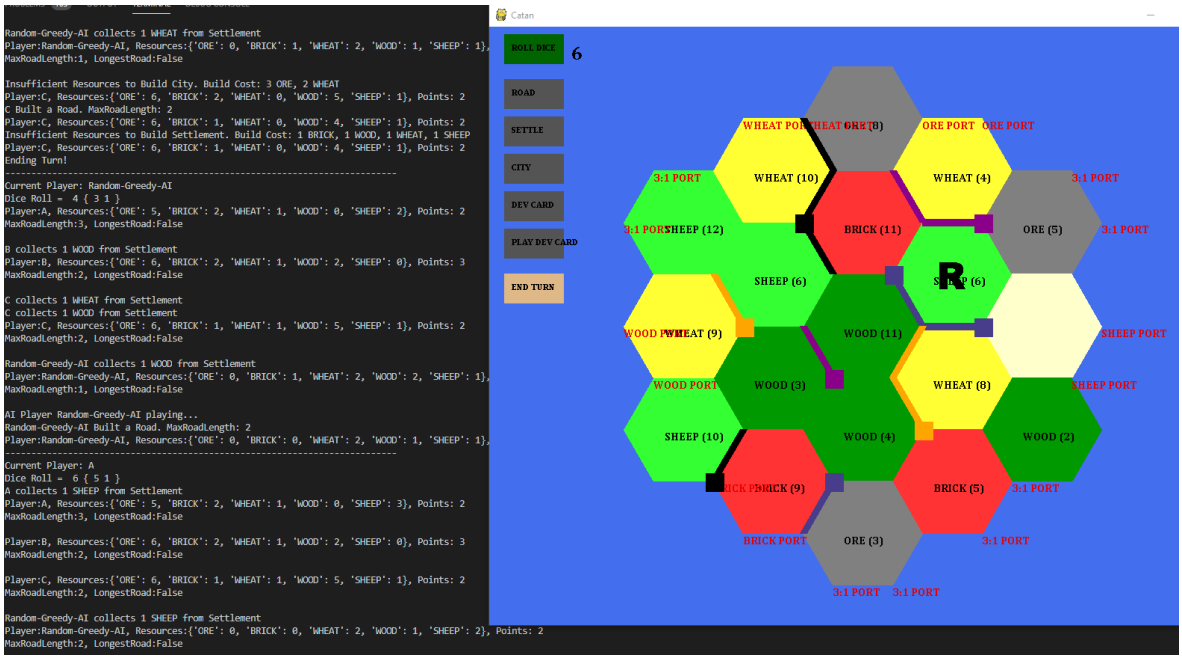
Slika 2.3 *Sadržaj postojeće implementacije agenta iz klijenta JSettlers*

S obzirom na fokus na igranje kroz sučelje, donesen je zaključak da bi ovaj radni okvir iziskivao previše promjena kako bi se osposobilo pokretanje igri bez sučelja i ljudskih igrača uz prikupljanje statistike. Također, predviđeni način implementacije novog agenta podrazumijeva zasebno nadjačavanje pojedinih postojećih strategija, što uvelike otežava metode koje se oslanjaju na jednu univerzalnu strategiju za odabir poteza, poput neuronske mreže. Sve u svemu, implementacije agenta i same igre u radnom okviru JSettlers pretjerano su složene za njihovo proučavanje, adaptaciju i nadogradnju u okvirima ovog rada.

2.2. CatanAI

Radni okvir CatanAI razvio je Karan Vombatkere 2020. godine u programskom jeziku Python[11]. Projekt je doživio nekoliko ažuriranja, no nisu redovna i ne unose značajne promjene u implementaciju klijenta, stoga je trenutna funkcionalnost vjerojatno i konačna. Izvorni kôd radnog okvira nalazi se u GitHub repozitoriju [8]. Za proučavanje ovog radnog okvira korišteno je razvojno okruženje PyCharm[12] tvrtke JetBrains.

CatanAI implementira vrlo jednostavno grafičko sučelje potpomognuto unosom i ispisom kroz naredbeno sučelje (konzolu) (vidi Slika 2.4). Za generiranje grafičkog sučelja s prikazom ploče koristi se biblioteka pygame[13], i njime se upravljaju temeljne funkcionalnosti poput bacanja kockica i gradnje, a kroz konzolu se provode interakcije poput odabira resursa za odbacivanje ili trgovanje. Implementacija igre podržava sve funkcionalnosti osnovnog Catana, no neke su značajno ograničene. Primjerice, igrač koji pokreće domaću trgovinu ima potpunu kontrolu nad istom, odnosno drugi igrač ju ne može odbiti ako je odabrana razmjena moguća.



Slika 2.4 *Sučelje klijenta CatanAI* [8]

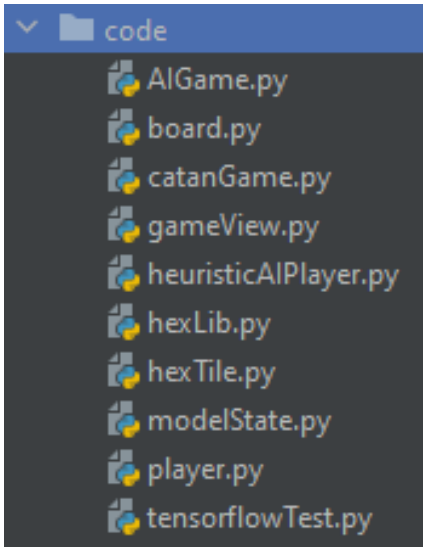
[11] https://www.python.org/

[12] https://www.jetbrains.com/pycharm/

[13] https://www.pygame.org/news

U projektu postoji implementacija ljudskog igrača (`player`), te je iz njega izveden model jednog funkcionalnog agenta (`heuristicAIPlayer`). Agent je strukturiran tako da se osnovni potez sastoji od slijednih pokušaja izgradnje naselja, grada, nekoliko cesti, i kupovanja razvojne karte. Za specifične tipove akcija poput micanja razbojnika postoje zasebne metode koje jednostavnom heuristikom biraju najbolji potez.

Nažalost, kôd igre je specifično prilagođen pokretanju samo s postojećim igračima – ljudskim igračem i heurističkim agentom. Za uvođenje novih agenata, trenutni kôd bi bilo potrebno značajno generalizirati i urediti. Osim toga, u radnom okviru nedostaje infrastruktura potrebna za prikupljanje statistike iz igri bez ljudskih igrača. Konačno, od svih analiziranih radnih okvira, CatanAI nudi najlošije korisničko iskustvo s kompleksnom interakcijom kroz konzolu. Iako projekt postiže dovoljnu razinu funkcionalnosti s relativno kratkim i jednostavnim kôdom (vidi Slika 2.5), te nije prezahtjevan za daljnje dorade, opseg dorada potrebnih prije „korisne“ implementacije prevelik je za ovaj rad.



Slika 2.5 *Sadržaj radnog okvira CatanAI*

2.3. Catanatron

Catanatron[14] je posljednji radni okvir za igru Catan analiziran u sklopu ovog rada. Razvio ga je Bryan Collazo 2021. godine u programskom jeziku Python. Projekt se još uvijek periodički ažurira raznim optimizacijama, poboljšanjima i popravcima. Inspiracija iza projekta i ciljevi razvoja Catanatrona objašnjeni su u članku *5 Ways NOT to Build a Catan AI* [9], kao i rezultati provedenog istraživanja s različitim implementacijama agenata. Izvorni kôd radnog okvira nalazi se u GitHub repozitoriju [10], a analiza funkcionalnosti u sklopu ovog rada provedena je korištenjem razvojnog okruženja PyCharm tvrtke JetBrains.

Catanatron je razvijen upravo s ciljem testiranja i usporedne raznih implementacija agenata za društvenu igru Catan, od jednostavnih *baseline* modela do onih koji se oslanjaju na metode iz domene strojnog učenja. Zato je postojeći kôd vrlo dobro strukturiran i generaliziran, te posjeduje infrastrukturu za prikupljanje statistike s mogućnostima formatiranog ispisa u konzolu, datoteku ili bazu podataka (vidi Slika 2.7). Pored osnovne logike za igru, radni okvir koristi strukturu web aplikacije za prikaz grafičkog sučelja (vidi Slika 2.6). *Front end* web aplikacije razvijen je pomoću biblioteke React[15] za programski jezik JavaScript[16], a *back end* poslužitelj koristi razvojni okvir Flask[17] za Python, te se spaja na PostgreSQL[18] bazu podataka. Sučelje mogu koristiti ljudski igrači, a može služiti i za prikazivanje igri samo s agentima. Za jednostavniji proces postavljanja web aplikacije, radni okvir koristi alat Docker Compose[19] za pokretanje svih potrebnih komponenti u zasebnim „kontejnerima“ (engl. *containers*).

---

[14] https://www.catanatron.com/

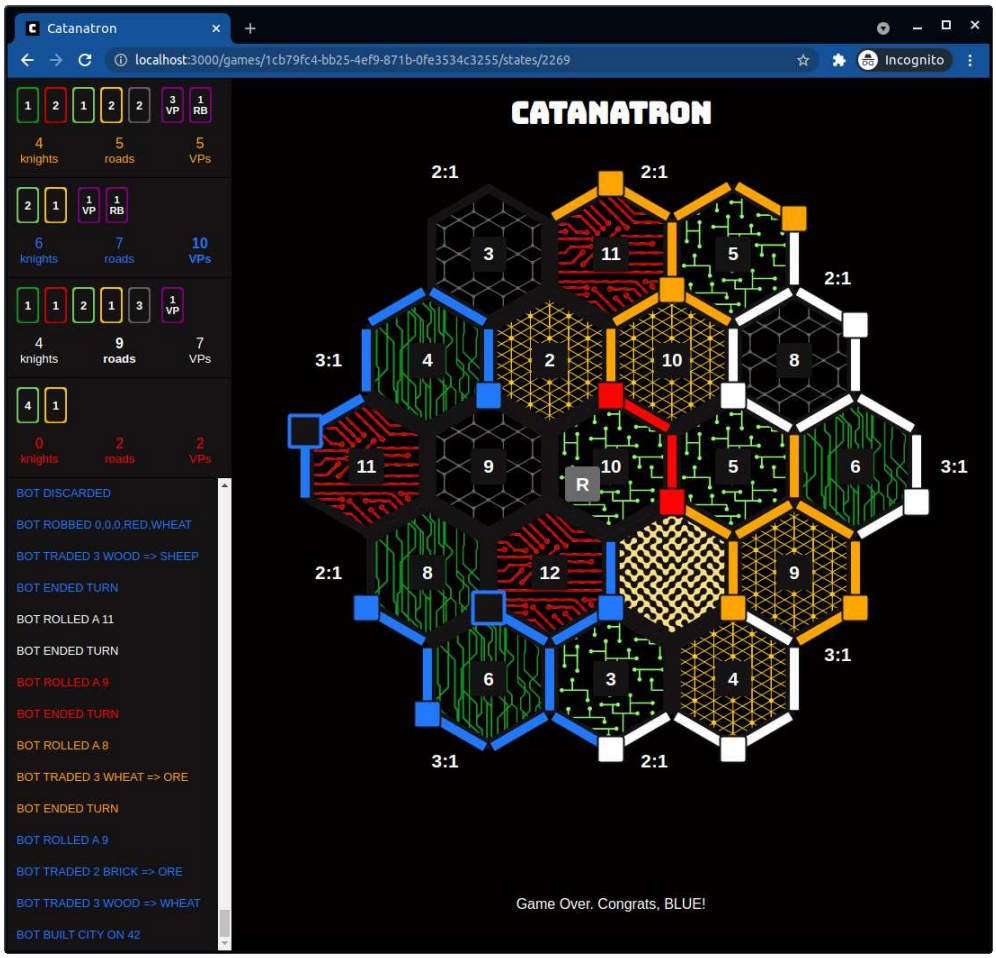
[15] https://reactjs.org/

[16] https://www.javascript.com/

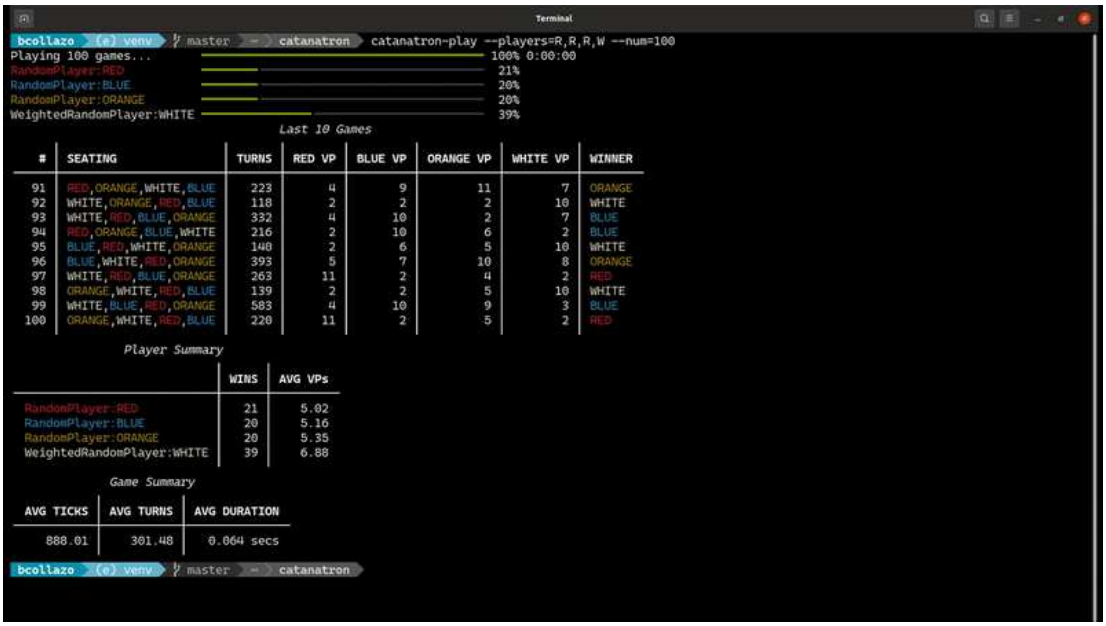
[17] https://flask.palletsprojects.com/en/2.1.x/

[18] https://www.postgresql.org/

[19] https://docs.docker.com/compose/



Slika 2.6 *Korisničko sučelje klijenta Catanatron* [9]



#	SEATING	TURNS	RED VP	BLUE VP	ORANGE VP	WHITE VP	WINNER
91	RED,ORANGE,WHITE,BLUE	223	4	9	11	7	ORANGE
92	WHITE,ORANGE,RED,BLUE	118	2	2	2	10	WHITE
93	WHITE,RED,BLUE,ORANGE	332	4	10	2	7	BLUE
94	RED,ORANGE,BLUE,WHITE	216	2	10	6	2	BLUE
95	BLUE,RED,WHITE,ORANGE	140	2	6	5	10	WHITE
96	BLUE,WHITE,RED,ORANGE	393	5	7	10	8	ORANGE
97	WHITE,RED,BLUE,ORANGE	263	11	2	4	2	RED
98	ORANGE,WHITE,RED,BLUE	139	2	2	5	10	WHITE
99	WHITE,BLUE,RED,ORANGE	583	4	10	9	3	BLUE
100	ORANGE,WHITE,RED,BLUE	220	11	2	5	2	RED

	WINS	AVG VPs
RandomPlayer:RED	21	5.02
RandomPlayer:BLUE	20	5.16
RandomPlayer:ORANGE	20	5.35
WeightedRandomPlayer:WHITE	39	6.88

AVG TICKS	AVG TURNS	AVG DURATION
888.01	301.48	0.064 secs

Slika 2.7 *Prikaz statistike za 100 odigranih igri s odabranim agentima u radnom okviru Catanatron* [10]

U sklopu Catanatrona postoji nekoliko funkcionalnih implementacija agenata za koje su u članku *5 Ways NOT to Build a Catan AI* [9] opisani postignuti rezultati međusobnog testiranja. Svi agenti nasljeđuju zajednički generalizirani model igrača (`Player`), te je on jedino programsko sučelje za koje implementacija igre mora znati. Model igrača temelji se na samo jednoj metodi koja se poziva za vrijeme njegovog poteza – `decide` (vidi Kôd 2.1). U ovu metodu se šalje trenutno stanje igre i lista svih akcija koje igrač može odigrati. Metoda `decide` će se ponovno pozivati za vrijeme trajanja trenutnog poteza, sve dok igrač ne odabere akciju za kraj poteza. Ovaj pristup pruža veliku razinu fleksibilnosti kod razvoja novih agenata s različitim strategijama za odabir akcija.

```python
class Player:
    """Interface to represent a player's decision logic.

    Formulated as a class (instead of a function) so that players
    can have an initialization that can later be serialized to
    the database via pickle.
    """

    def __init__(self, color, is_bot=True):
        self.color = color
        self.is_bot = is_bot

    def decide(self, game, playable_actions):
        """Should return one of the playable_actions.

        Args:
            game (Game): complete game state. read-only.
            playable_actions (Iterable[Action]): options right now
        """
        raise NotImplementedError
```

Kôd 2.1 *Osnovni model igrača radnog okvira Catanatron*

Glavni nedostaci ovog radnog okvira su nefunkcionalnost domaće trgovine među igračima, te nekolicina problema u interakciji ljudskog igrača sa sučeljem, poput nasumičnog odbacivanja karti i micanja razbojnika, ili nemogućnosti igranja razvojnih karata. Ipak, s obzirom na vrlo dobru strukturu izvornog kôda i infrastrukturu za testiranje agenata, kvalitetno grafičko sučelje, te značajno optimiziran model igre i igrača, radni okvir Catanatron odabran je za razvoj programskog rješenja u sklopu ovog rada.

3. Implementacija

Programsko rješenje ovog rada razvijeno je u već spomenutom razvojnom okruženju PyCharm (verzija 2021.3.2) tvrtke JetBrains. Strukturirano je kao zaseban modul pod imenom `catan_ai` u sklopu postojeće strukture projekta Catanatron, koja već uključuje module `catanatron_core`, `catanatron_experimental`, `catanatron_gym`, `catanatron_server` i `ui`. Kao i ostali moduli projekta Catanatron, `catan_ai` je prilagođen za instalaciju alatom za upravljanje Python paketima pip[20].

Implementacija modula podijeljena je na pakete `game`, `players`, `training` i `util`, te dodatno sadrži direktorij `resources`. U paketu `players` nalaze se skripte s implementacijama svih vrsta igrača kreiranih i analiziranih u sklopu ovog projekta, te skripte s bilo kakvim pomoćnim podacima za iste. Implementirana su tri polazna (engl. *baseline*) igrača: Random Player, Weighted Random Player i Greedy VP Player, te tri napredna igrača: Value Function Player, Neural Network Player i Tree Search Player. Polazni igrači osmišljeni su za potrebe upoznavanja s radom u radnom okviru Catanatron, i analize uspješnosti naprednih igrača. U nastavku su opisani i obrazloženi modeli i karakteristike svih navedenih igrača.

3.1. Polazni igrači

Prvi od takozvanih *baseline* igrača je Random Player. Kako i njegov naziv upućuje, ovaj igrač nasumično bira jednu od raspoloživih akcija svaki put kada je pozvan da povuče svoj potez. Za potrebe nasumičnog odabira iz liste mogućih akcija, koristi se metoda `random.choice` (vidi Kôd 3.1). S obzirom na nepredvidivu i besciljnu prirodu ovog igrača, moguće je da igre s njim traju izuzetno dugo, no Catanatron implementira ograničenje igre na tisuću krugova, nakon čega zaustavlja igru bez pobjednika.

[20] https://pypi.org/project/pip/

```
class RandomPlayer(Player):

    def decide(self, game, playable_actions):
        return random.choice(playable_actions)
```

Kôd 3.1 *Implementacija igrača Random Player*

Sljedeći *baseline* igrač je Weighted Random Player. Ovaj igrač se također oslanja na nasumični odabir, no, za razliku od Random Playera, koristi vrlo pojednostavljeno poznavanje ciljeva igre Catan za dodjelu težina različitim vrstama akcija. Tako, primjerice, izgradnji gradova dodjeljuje težinsku vrijednost 1000, izgradnji naselja 700, izgradnji cesta 300, i tako dalje. Svi neizdvojeni tipovi akcija ulaze u izbor metode `random.choice` s težinom 1. U isječku (Kôd 3.2) mogu se vidjeti sve definirane težine za izdvojene vrste akcija, te implementacija koja ih koristi.

```
class WeightedRandomPlayer(Player):

    def __init__(self, color, is_bot=True):
        super().__init__(color, is_bot)

        self.action_type_weights = {
            ActionType.BUILD_CITY: 1000,
            ActionType.BUILD_SETTLEMENT: 700,
            ActionType.BUILD_ROAD: 300,
            ActionType.BUY_DEVELOPMENT_CARD: 250,
            ActionType.PLAY_KNIGHT_CARD: 50,
            ActionType.PLAY_MONOPOLY: 50,
            ActionType.PLAY_ROAD_BUILDING: 50,
            ActionType.PLAY_YEAR_OF_PLENTY: 50
        }

    def decide(self, game, playable_actions):
        if len(playable_actions) == 1:
            return playable_actions[0]

        weights = list(map(lambda action:
                           self.action_type_weights.get(action.action_type, 1),
                           playable_actions
                           ))

        return random.choices(playable_actions, weights, k=1)[0]
```

Kôd 3.2 *Implementacija igrača Weighted Random Player*

Posljednji *baseline* model igrača je Greedy VP Player. Njegov odabir akcije se u potpunosti temelji na „pohlepnom“ maksimiziranju svog broja pobjedničkih bodova (VP). Ovakve strategije nazivaju se „pohlepnima“ jer ne uzimaju u obzir mogućnost da neka manje profitabilna početna akcija može dovesti do kasnije nagrade, već samo gledaju isplativost sljedećeg poteza. Odličan primjer onoga što ovakav igrač propušta u kontekstu Catana je nagrada za najdužu cestu ili najveću vojsku, koje se mogu dobiti tek nakon nekoliko uzastopnih „neisplativih“ akcija. Greedy VP Player će provjeriti isplativost pojedine akcije tako da ju proba odigrati na kopiji trenutnog stanja igre, i usporedi svoj broj bodova nakon svake isprobane akcije. Ako više akcija rezultira istim maksimalnim brojem bodova, igrač će nasumično odabrati jednu od njih (vidi Kôd 3.3). Razlog zašto nije uvijek moguće unaprijed pretpostaviti koliko bodova će neka akcija donijeti je upravo zbog bodova koji se dobivaju pod dodatnim uvjetima, a ne zbog same akcije, poput nagrade za najdužu cestu i najveću vojsku.

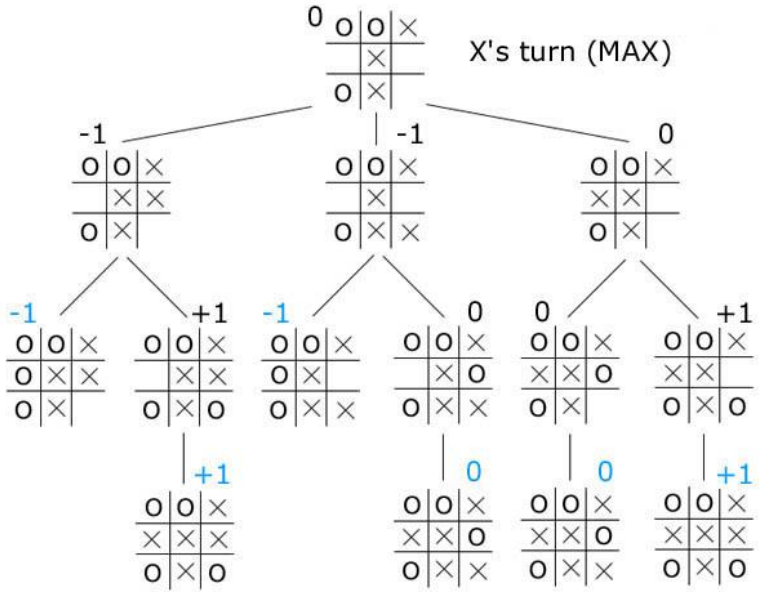
```
class GreedyVpPlayer(Player):

    def decide(self, game, playable_actions):
        if len(playable_actions) == 1:
            return playable_actions[0]

        best_vp = 0
        best_actions = []

        for action in playable_actions:
            game_copy = game.copy()
            game_copy.execute(action)

            state = game_copy.state
            current_vp = get_actual_victory_points(state, self.color)

            if current_vp > best_vp:
                best_vp = current_vp
                best_actions = [action]
            elif current_vp == best_vp:
                best_actions.append(action)

        return random.choice(best_actions)
```

Kôd 3.3 *Implementacija igrača Greedy VP Player*

3.2. Value Function Player

Value Function Player je prvi pokušaj izrade naprednog agenta za društvenu igru Catan u ovom radu. Za evaluaciju isplativosti sljedećeg poteza koristi sličan pristup kao i Greedy VP Player, ali, umjesto maksimiziranja isključivo broja pobjedničkih bodova, maksimizira vrijednost prilagođene linearne heurističke funkcije.

Heuristika je tehnika koja se koristi u rješavanju problema pretraživanja stanja. U problemima pretraživanja stanja želimo naći slijed akcija koji nas dovodi od početnog do ciljnog stanja. U našem slučaju, početno stanje je trenutno stanje igre, a ciljno stanje je pobjeda. Osnovni pristup pronalasku puta od početka do cilja je potpuno pretraživanje prostora stanja (engl. *brute-force search*), no u društvenim igrama poput Catana, šaha i sličnih, to nije jednostavno. Prostor stanja igre može se prikazati u obliku stabla, gdje je svaki čvor jedno stanje, i grana se u nova stanja za svaki mogući potez igrača (vidi Slika 3.1). Iz toga se može primijetiti da će prostor stanja rasti eksponencijalno, stoga je potrebno optimizirati pretraživanje, a to se može postići heuristikom. Heuristika obuhvaća pravila o prirodi problema kojima pretraživanje pokušavamo usmjeriti brže prema cilju [11]. U slučaju Catana, moguće je definirati heurističku funkciju koja procjenjuje koliko je određeno stanje povoljno za igrača na temelju poznatih podataka poput broja izgrađenih gradova, naselja i cesta, proizvodnje resursa kojom raspolaže, i sličnog. Value Function Player na svojim potezima evaluira upravo takvu funkciju, i na temelju njenih vrijednosti u stanjima nakon svake moguće akcije, bira najbolji potez.



Slika 3.1 *Stablo igre križić-kružić* [12]

U heurističku funkciju Value Function Playera ulazi 18 vrijednosti agregiranih iz skupa od 48 sirovih podataka o trenutnom stanju. Prilikom agregiranja i izračuna konačne vrijednosti funkcije, koristi se 25 parametara kojima se različiti podaci množe i potom sumiraju. Popis svih parametara i njihovih početnih vrijednosti može se vidjeti u isječcima (Kôd 3.4 i Kôd 3.5). Parametri su podijeljeni u tri skupine: multiplikatori troškova, multiplikatori resursa i multiplikatori vrijednosti. Multiplikatori troškova koriste se za izračun trenutne vrijednosti resursa u ruci igrača, i u suštini predstavljaju koliko je poželjno štedjeti resurse za pojedini trošak. Troškovi naravno uključuju izgradnju grada, naselja i ceste, te kupovinu razvojne karte (vidi Kôd 3.6). Multiplikatori resursa služe za evaluaciju igračeve proizvodnje resursa. Ukupna vjerojatnost da igrač dobije pojedini resurs ovisi zbroju vjerojatnosti da odgovarajuća pločica terena na kojoj igrač ima naselje bude odabrana za proizvodnju bacanjem kockica. Te vjerojatnosti se množe navedenim multiplikatorima. Preostali multiplikatori vrijednosti su manje složeni, i pretežno se odnose na pojedinačne podatke iz trenutnog stanja. Dodatno ćemo istaknuti multiplikator `RESOURCE_TYPES` kojim se nagrađuje broj različitih vrsta resursa koje je igrač pokrio proizvodnjom. Bit će maksimalno nagrađen ako je svojim gradovima i naseljima pokrio proizvodnju svih 5 resursa. Također, multiplikator `PORT_VALUE` množi vrijednost funkcije koja računa koliko je vjerojatno da će igrač proizvesti resurs za kojeg posjeduje luku. Time je ugrađena preferencija prema zauzimanju luke u kojoj se razmjenjuje resurs kojeg igrač očekuje često dobivati (vidi Kôd 3.7). Među parametrima postoji nekolicina koje evaluiramo negativno. Primjerice, `DISCARD_PENALTY` osigurava da igrač ne pretjeruje s brojem karata u ruci kako ih ne bi morao odbaciti kada je bačen broj 7. Ostali negativni parametri odnose se na prednost koju imaju igračevi suparnici. Korisni su za odabir optimalne pozicije za pomicanje razbojnika, „krađu“ potencijalnih lokacija za izgradnju, te za napredniju verziju ovog igrača – Tree Search Player.

```
self.expense_multipliers: dict[Expense, int] = {
    Expense.CITY: 100,
    Expense.SETTLEMENT: 50,
    Expense.ROAD: 20,
    Expense.DEV_CARD: 20
}

self.resource_multipliers: dict[Resource, int] = {
    Resource.BRICK: 100,
    Resource.ORE: 100,
    Resource.SHEEP: 100,
    Resource.WHEAT: 100,
    Resource.WOOD: 100
}
```

Kôd 3.4 *Parametri za vrijednost resursa u ruci i proizvodnje resursa*

```
self.value_multipliers: dict[Value, int] = {
    Value.VP: 1500,

    Value.CITIES: 400,
    Value.SETTLEMENTS: 250,
    Value.ROAD_LENGTH: 150,
    Value.KNIGHTS: 80,
    Value.DEV_CARDS: 40,

    Value.RESOURCE_TYPES: 120,

    Value.PORT_VALUE: 100,

    Value.BUILDABLE_NODES: 120,
    Value.BUILDABLE_EDGES: 80,

    Value.DISCARD_PENALTY: -100,

    Value.ENEMY_VP: -200,

    Value.ENEMY_RESOURCE_PRODUCTION: -100,
    Value.ENEMY_RESOURCE_TYPES: -50,

    Value.ENEMY_BUILDABLE_NODES: -100,
    Value.ENEMY_BUILDABLE_EDGES: -50
}
```

Kôd 3.5 *Parametri za ostale vrijednosti stanja*

```
def _evaluate_hand(self, state):
    value = 0

    hand = player_deck_to_array(state, self.color)
    resource_count = Counter(hand)

    for expense, cost in expense_costs.items():
        current_value = 0
        for resource in cost.keys():
            num_resource = resource_count.get(resource.name, 0)
            current_value += min(1, num_resource / cost[resource])
        value += current_value * self.expense_multipliers[expense]

    return value
```

Kôd 3.6 *Funkcija za evaluaciju resursa u ruci*

```
def _evaluate_ports(self, state, production):
    value = 0

    ports = set(state.board.get_player_port_resources(self.color))
    for port in ports:
        if port is None:
            value += sum(production.values()) / len(production.values())
        else:
            value += production[port]

    return value * self.value_multipliers[Value.PORT_VALUE]
```

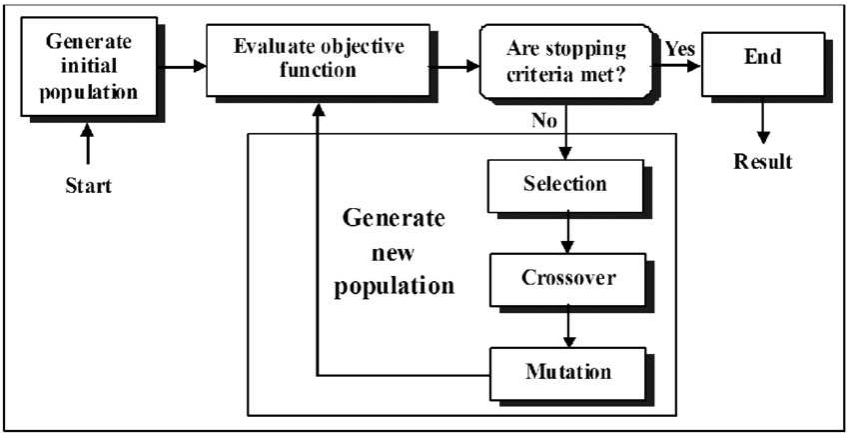
Kôd 3.7 *Funkcija za evaluaciju luka u posjedu*

3.2.1. Učenje parametara

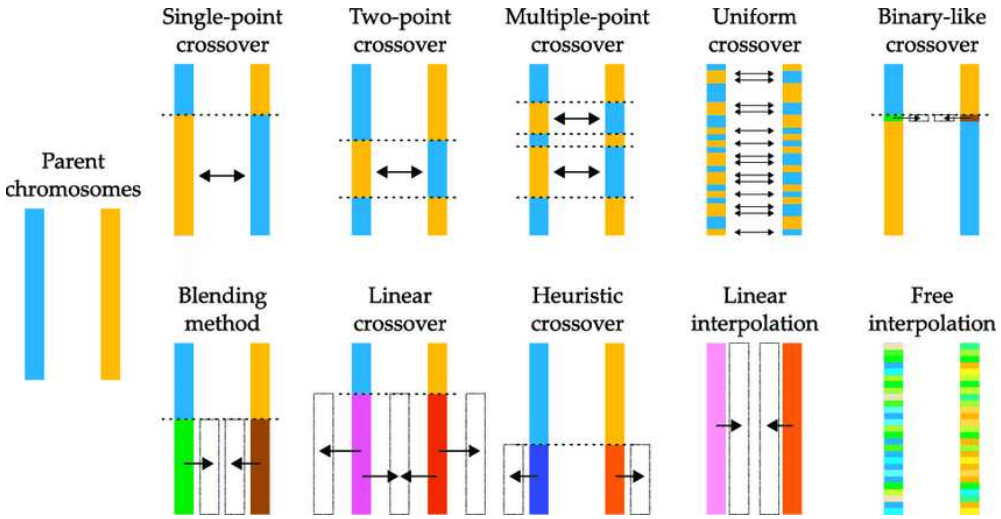
S obzirom na to da se ponašanje Value Function Playera može bitno promijeniti ovisno o vrijednostima opisanih parametara heurističke funkcije, postoji prilika da se primijeni neka metoda strojnog učenja kako bi se parametri optimizirali. Najpogodniji način da saznamo koji od igrača ima bolju kombinaciju parametara je da se sukobe u nizu od nekoliko igri, i pobjednika koristimo kao primjer za daljnju optimizaciju. Oko ove ideje se relativno lako može izgraditi genetski algoritam s turnirskom selekcijom.

Genetski algoritam je algoritam za pronalazak prihvatljivog rješenja nekog problema koji se temelji na prirodnoj evoluciji. Zato spada u skupinu takozvanih evolucijskih algoritama. Za provođenje genetskog algoritma, prvo se stvara populacija različitih jedinki. Svaka jedinka predstavlja jedno moguće rješenje problema koje ima svoju dobrotu (engl. *fitness*). Vraćajući se na prirodnu evoluciju, može se reći da jedinka predstavlja jedan kromosom. U našem slučaju, jedinka će biti jedna moguća kombinacija parametara Value Function Playera. Genetski algoritam napreduje tako da iterativno kreiramo nove generacije populacije, s ciljem da se približavaju optimalnom rješenju problema (vidi Slika 3.2). Način na koji se to postiže je selekcija. Cilj selekcije je odabrati bolje jedinke iz populacije, odnosno jedinke s većom dobrotom, te pomoću njih u sljedećoj generaciji zamijeniti lošije. U implementiranom genetskom algoritmu koristi se turnirska selekcija. Turnirska selekcija provodi se nad nasumičnim uzorcima populacije. Kako je Catan igra za do 4 igrača, provodit će se turnirska selekcija nad 4 jedinke. Dobrote odabranih jedinki uspoređujemo igranjem proizvoljnog broja partija Catana i brojanjem njihovih pobjeda. Dvije bolje jedinke bit će „roditelji“ od kojih se križanjem stvaraju nove jedinke koje će

zamijeniti gubitnike turnira, tj. 3. i 4. igrača. Križanjem se osigurava da „djeca“ nasljeđuju kombinaciju svojstava oba roditelja, kako bi se nastavila potraga za boljim rješenjima. S obzirom na to da optimiziramo brojčane parametre na kontinuiranom području, jedno dijete stvaramo aritmetičkim križanjem, a drugo heurističkim (vidi Kôd 3.8). Područja u kojima se ovim križanjima mogu naći vrijednosti parametara djece su između vrijednosti dvaju roditelja, ili „preko“ vrijednosti boljeg roditelja (vidi Slika 3.3). Oba područja su obećavajuće mjesto za traženje boljeg rješenja. Kako bi se spriječilo zapinjanje u lokalnim optimumima problema, u genetskim algoritmima može se provoditi mutacija nad novostvorenim jedinkama. U ovoj implementaciji se mutacija i raznolikost početne populacije osiguravaju dodavanjem nasumičnih vrijednosti iz normalne raspodjele sa srednjom vrijednosti 0, proizvoljnom standardnom devijacijom, i proizvoljnom vjerojatnošću (vidi Kôd 3.9).



Slika 3.2 *Koraci genetskog algoritma* [13]



Slika 3.3 *Primjeri različitih metoda križanja* [14]

```python
@staticmethod
def arithmetic_crossover(unit1: tuple[int | float, ...], unit2: tuple[int | float, ...]) -> tuple[int | float, ...]:
    child = []

    for i in range(len(unit1)):
        alpha = random.uniform(0, 1)
        value = alpha * unit1[i] + (1 - alpha) * unit2[i]
        if isinstance(unit1[i], int):
            value = int(value)
        child.append(value)

    return tuple(child)

@staticmethod
def heuristic_crossover(unit1: tuple[int | float, ...], unit2: tuple[int | float, ...]) -> tuple[int | float, ...]:
    child = []

    for i in range(len(unit1)):
        alpha = random.uniform(0, 1)
        value = alpha * (unit1[i] - unit2[i]) + unit1[i]
        if isinstance(unit1[i], int):
            value = int(value)
        child.append(value)

    return tuple(child)
```

Kôd 3.8 *Implementacije aritmetičkog i heurističkog križanja*

```python
@staticmethod
def mutate(unit: tuple[int | float, ...], mutation_chance: float, mutation_strength: int | float)\
        -> tuple[int | float, ...]:
    unit = list(unit)

    for i in range(len(unit)):
        if random.random() < mutation_chance:
            value = random.gauss(0, mutation_strength)
            if isinstance(unit[i], int):
                value = int(value)
            unit[i] += value

    return tuple(unit)
```

Kôd 3.9 *Implementacija mutacije*

Važno je napomenuti da ova primjena genetskog algoritma nije vrlo precizna. Nije garantirano da će nedvojbeno bolja jedinka uvijek pobijediti, jer Catan ima određene nedeterminističke karakteristike, poput bacanja kockica, izvlačenja razvojnih karata, i samog rasporeda ploče. Drugim riječima, uspjeh igrača jednim manjim dijelom ovisi o pukoj sreći. No nije nam ni cilj naći jedno optimalno rješenje matematičkog problema, već bar u prosjeku pomaknuti sposobnosti Value Function Playera prema boljem od njegovih početnih parametara. Opisana implementacija genetskog algoritma je za to sasvim sposobna.

U isječcima (Kôd 3.10 i Kôd 3.11) prikazane su vrijednosti parametara Value Function Playera nakon 200 iteracija opisanog genetskog algoritma s populacijom od 60 jedinki, 20 turnira po iteraciji, i 20 odigranih partija po turniru. U komentarima kôda zabilježen je iznos razlike naučene vrijednosti u odnosu na početnu.

```
self.expense_multipliers: dict[Expense, int] = {
    Expense.CITY: 298,  # +198
    Expense.SETTLEMENT: 13,  # -37
    Expense.ROAD: -71,  # -91
    Expense.DEV_CARD: -24  # -44
}

self.resource_multipliers: dict[Resource, int] = {
    Resource.BRICK: 313,  # +213
    Resource.ORE: 424,  # +324
    Resource.SHEEP: 158,  # +58
    Resource.WHEAT: 369,  # +269
    Resource.WOOD: 430  # +330
}
```

Kôd 3.10 *Parametri za vrijednost resursa u ruci i proizvodnje resursa nakon učenja genetskim algoritmom*

```
self.value_multipliers: dict[Value, int] = {
    Value.VP: 1855,  # +355

    Value.CITIES: 1790,  # +1390
    Value.SETTLEMENTS: 295,  # +45
    Value.ROAD_LENGTH: 494,  # +344
    Value.KNIGHTS: 215,  # +135
    Value.DEV_CARDS: -1136,  # -1176

    Value.RESOURCE_TYPES: 428,  # +308

    Value.PORT_VALUE: -34,  # -134

    Value.BUILDABLE_NODES: 314,  # +194
    Value.BUILDABLE_EDGES: 24,  # -56

    Value.DISCARD_PENALTY: -63,  # +37

    Value.ENEMY_VP: 64,  # +264

    Value.ENEMY_RESOURCE_PRODUCTION: 232,  # +332
    Value.ENEMY_RESOURCE_TYPES: -140,  # -90

    Value.ENEMY_BUILDABLE_NODES: -41,  # +59
    Value.ENEMY_BUILDABLE_EDGES: 717  # +767
}
```

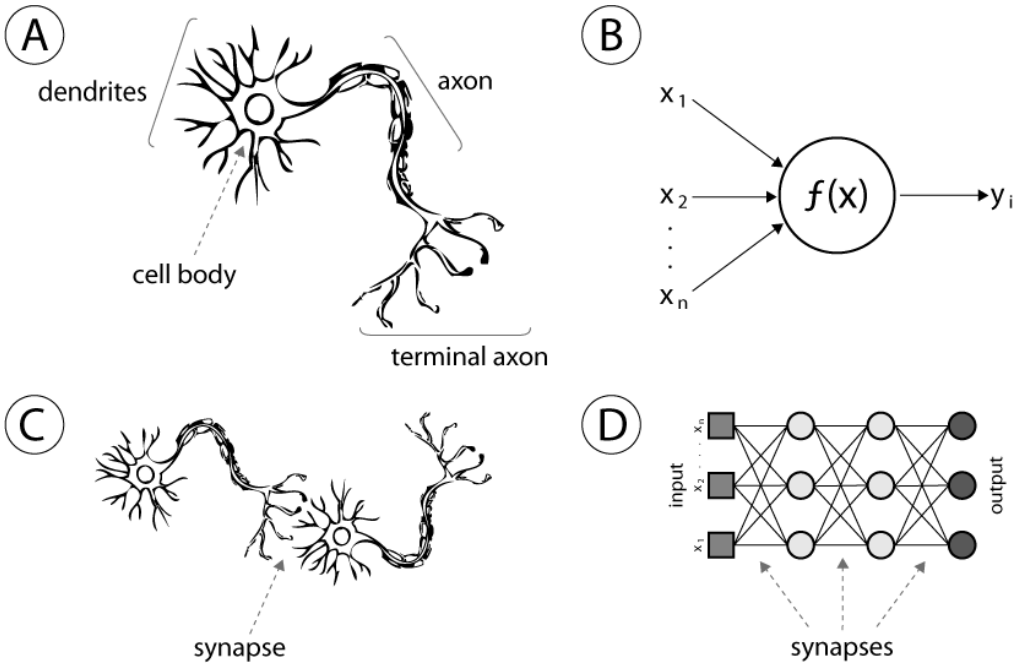
Kôd 3.11 *Parametri za ostale vrijednosti stanja nakon učenja genetskim algoritmom*

3.3. Neural Network Player

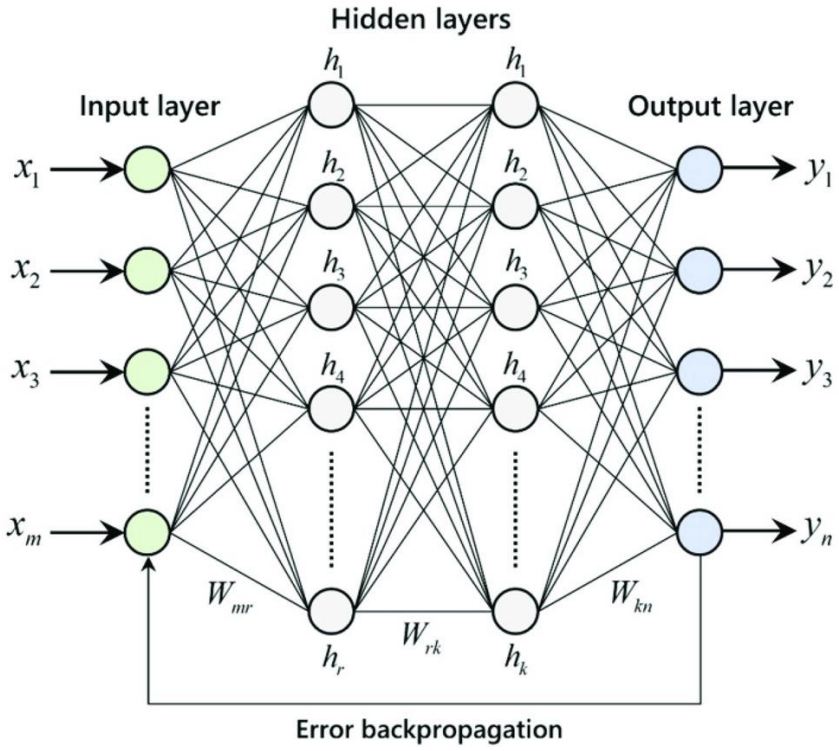
Kod agenta Neural Network Player, ideja je bila provoditi proces evaluacije stanja igre neuronskom mrežom umjesto linearnom heurističkom funkcijom. Cilj ovog pristupa je omogućiti da se učenjem neuronske mreže otkriju još neke korelacije između podataka iz trenutnog stanja koje nisu nužno predviđene heuristikom Value Function Playera.

Umjetne neuronske mreže (engl. *artificial neural network*, ANN) su još jedan primjer programskih rješenja koja se ugledaju na prirodni svijet. Proizašle su iz ideje da bi model

umjetne inteligencije (engl. *artificial intelligence*, AI) koji uspješno aproksimira rad mozga inherentno bio inteligentan. Mozak odnosno neuronska mreža se sastoji od velikog broja neurona, gdje svaki radi vrlo jednostavnu obradu podataka primljenih od brojnih drugih neurona, rezultate šalje drugim neuronima, te to mogu u određenoj mjeri raditi paralelno (vidi Slika 3.4). Klasičnu umjetnu neuronsku mrežu dijelimo na slojeve neurona (engl. *layers*). Mreža se sastoji od ulaznog sloja, gdje svaki neuron predstavlja jednu vrijednost ulaznog niza podataka, proizvoljnog broja skrivenih slojeva, gdje neuroni vrše nelinearne transformacije nad linearnom kombinacijom svojih ulaza, te izlaznog sloja koji vraćaju izračunate izlazne podatke mreže. Utjecaj pojedine ulazne vrijednosti na neuron ovisi o težini tog ulaza. Za neuronske mreže, te težine su glavni parametri koji se optimiziraju da bi se prilagodio izlaz mreže. Učenje težina provodi se algoritmom rasprostiranja unatrag (engl. *backpropagation*), na temelju odstupanja od očekivanih vrijednosti na izlazu (funkcija gubitka). Njime računamo doprinos pojedinih neurona izlaznoj pogrešci, sloj po sloj, počevši od izlaza, te korigiramo njihove težine kako bismo smanjili grešku (vidi Slika 3.5).



Slika 3.4 *Usporedba prirodnih i umjetnih neurona* [15]



Slika 3.5 *Prikaz umjetne neuronske mreže* [16]

Za neuronsku mrežu agenta Neural Network Player, na ulazu se koristi istih 48 podataka o stanju igre kao i kod heurističke funkcije Value Function Playera, te se vraća 1 izlaz – procijenjena vrijednost tog stanja. Sam model mreže implementiran je pomoću biblioteke TensorFlow[21]. Riječ je o sekvencijalnom modelu s proizvoljnim brojem potpuno povezanih skrivenih slojeva neurona. Dok se prilikom treniranja mreže koristi puna snaga paketa TensorFlow, za pohranu modela u objekt igrača mora se koristiti model TensorFlow Lite (vidi Kôd 3.12). Ova vrsta modela je prilagođena za reprezentaciju već naučene mreže koje će se koristiti isključivo za generiranje novih predikcija, te se zato može pohraniti u pojednostavljenom obliku kao niz bajtova. Ovo pomaže u situacijama kada se objekt igrača, kao dio stanja igre, mora serijalizirati i spremati u datoteke ili bazu podataka, kao što je slučaj s igranjem kroz grafičko sučelje.

[21] https://www.tensorflow.org/

```
def create_model(self, dropout: float = 0):
    layers = [InputLayer(input_shape=NeuralNetworkPlayer.input_size)]

    initializer = RandomUniform(minval=-10, maxval=10)

    for size in self.layers:
        layers.append(Dense(size, activation='relu', kernel_initializer=initializer))
        if dropout > 0:
            layers.append(Dropout(dropout))

    layers.append(Dense(1, activation='linear', kernel_initializer=initializer))

    model = Sequential(layers=layers)

    return model

@staticmethod
def convert_model(model):
    converter = TFLiteKerasModelConverterV2(model)
    converter.optimizations = [Optimize.DEFAULT]
    return converter.convert()

@staticmethod
def create_interpreter(model):
    return Interpreter(model_content=model)
```

Kôd 3.12 *Funkcije za kreiranje modela TensorFlow i TensorFlow Lite*

3.3.1. Učenje mreže

U teoriji, umjesto algoritmom rasprostiranja unatrag, težine neuronske mreže može se optimizirati genetskim algoritmom kao i svaki drugi niz parametara, i taj pristup se naziva neuroevolucija. Nažalost, početne vrijednosti težina Neural Network Playera nije jednostavno prilagoditi da se izbjegne potpuno nasumično ponašanje netrenirane mreže. Zbog toga, u pokušajima pokretanja genetskog algoritma bez pred-treniranja dolazi do velikog broja nezavršenih igri, odnosno igri bez pobjednika u kojima je pređeno ograničenje od tisuću krugova. To dovodi do problema da genetski algoritam ne uspijeva konvergirati prema boljim rješenjima jako velik broj iteracija. Kako bi se zaobišao ovaj problem, odlučeno je da će se mreža pred-trenirati pomoću alata TensorFlow. Za ovo treniranje kreira se skup podataka (engl. *dataset*) koji detaljno opisuje ponašanje heurističke funkcije Value Function Playera. Neuronska mreža djelomično se pred-trenira prema navedenom skupu podataka. Treniranje se provodi uz optimizator Adam, s padajućom stopom učenja i proizvoljnom vjerojatnošću izostavljanja neurona (engl. *dropout*), kako bi se zadržala visoka razina generalizacije (vidi Kôd 3.13). Tijekom razvoja

isprobane su razne arhitekture mreža, te je utvrđeno da dva skrivena sloja dimenzija 32 i 16 dovode do dobrih rezultata s prihvatljivim brojem parametara.

Uz pred-treniranje nad skupom podataka i treniranje genetskim algoritmom, Neural Network Player se uspješno snalazi u igri i ostvaruje dobre rezultate.

```
print(f'Compiling tensorflow model...')
model = player.create_model(args.dropout)
learning_rate = ExponentialDecay(args.learning_rate, decay_steps=len(train), decay_rate=0.5, staircase=True)
optimizer = Adam(learning_rate=learning_rate)
model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mae'])
print(f'Input Shape: (None, {NeuralNetworkPlayer.input_size})')
model.summary()
print()

print(f'Fitting tensorflow model...\n')
callbacks = []
if args.early_stop_patience > 0:
    callbacks.append(EarlyStopping(monitor='val_mae', patience=args.early_stop_patience, min_delta=0.5))
history = model.fit(
    train, epochs=args.epochs, validation_data=valid, callbacks=callbacks,
    use_multiprocessing=True, workers=8
)
```

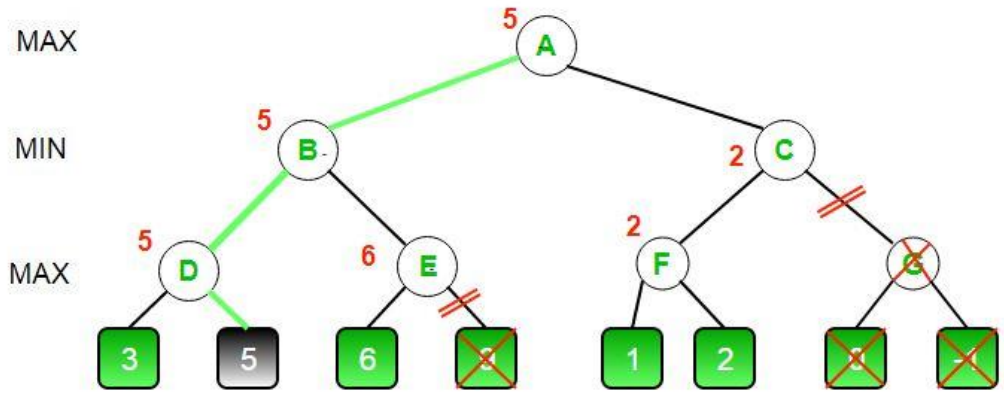
Kôd 3.13 *Postupak pred-treniranja mreže Neural Network Playera*

3.4. Tree Search Player

Tree Search Player je posljednji napredni agent implementiran u sklopu ovog rada. Uvelike se temelji na Value Function Playeru – koristi istu heurističku funkciju, no ovaj put za pravo pretraživanje stabla igre algoritmom minimaks (engl. *minimax*), do proizvoljne dubine.

Algoritmom minimaks igrač traži optimalan potez nastojeći maksimizirati svoj dobitak, uz pretpostavku da će ga njegov protivnik ili protivnici nastojati minimizirati. Time znamo odabrati vlastiti potez, i predvidjeti protivnikov. Korištenjem heuristike osigurava se da se algoritam ne mora provoditi do završnih stanja igre, kada se zna tko je pobjednik, već se pretraga može ograničiti na određenu dubinu gdje se dosegnuta stanja evaluiraju heurističkom funkcijom. Iako je dubina pretrage ograničena, heuristički pristup i dalje omogućuje praćenje određenih strategija kroz više poteza. Algoritam minimaks dodatno se može optimizirati uvođenjem alfa-beta podrezivanja (engl. *alpha-beta pruning*). Podrezivanje se koristi da se, na temelju heuristike, iz pretraživanja odbace

sljedovi poteza koji igrača ne mogu dovesti do povoljnijeg stanja od nekog već posjećenog. Na primjeru alfa-beta podrezivanja (Slika 3.6), MAX označava polazna stanja igrača koji nastoji maksimizirati svoj dobitak, a MIN njegovog protivnika koji mu nastoji minimizirati taj dobitak. Ispod čvora MIN vrši se alfa-podrezivanje – odbacuju se podstabla kada smo sigurni da MAX neće odabrati trenutni MIN čvor, a ispod čvora MAX vrši se beta-podrezivanje – odbacuju se podstabla kada smo sigurni da MIN neće odabrati trenutni MAX čvor.



Slika 3.6 *Minimaks pretraživanje uz alfa-beta podrezivanje* [17]

Tree Search Player implementira upravo minimaks pretraživanje s alfa-beta podrezivanjem (vidi Kôd 3.14). Bez obzira na podrezivanje, algoritam nije dovoljno efikasan na dubinama većim od 5 akcija. Glavni razlog za to je postojanje nedeterminističkih poteza u Catanu. Naime, kod bacanja kockica igračeva akcija zapravo rezultira s 11 različitih mogućih stanja za samo jedan odigran potez. Prilikom kupovanja razvojne karte, postoji 5 različitih vrsta karata koje igrač može izvući, a prilikom pomicanja razbojnika, postoji 5 različitih resursa koje bi igrač mogao ukrasti od protivnika. Navedene situacije, koje granaju stablo igre u više različitih stanja za samo jednu potencijalnu akciju, uvelike povećavaju porast broja stanja u stablu igre Catan, u odnosu na neke druge društvene igre. Također, u implementaciji je potrebno uzeti ove slučajeve u obzir i ručno ekspandirati svaki čvor koji sadrži nedeterminističku akciju na sve moguće ishode te akcije (vidi Kôd 3.15).

```
        if is_maximizing_player:
            parent.value = -math.inf
            best_nodes = []

            for action in parent.game.state.playable_actions:
                action_node = ActionNode(action)
                action_node.value = 0
                parent.children.append(action_node)

                state_nodes = self._expand_node(parent, action_node)
                for state_node in state_nodes:
                    result = self.search(state_node, depth - 1, alpha, beta)
                    action_node.value += result.value * state_node.probability

                if action_node.value > parent.value:
                    parent.value = action_node.value
                    best_nodes = [action_node]
                elif action_node.value == parent.value:
                    best_nodes.append(action_node)

                alpha = max(alpha, parent.value)

                if alpha >= beta:
                    break

            return random.choice(best_nodes)
```

Kôd 3.14 *Isječak algoritma minimaks (čvor MAX)*

```
def _expand_node(self, parent: StateNode, action_node: ActionNode) -> list[StateNode]:
    game = parent.game
    action = action_node.action

    if action.action_type == ActionType.ROLL:
        filtered_rolls = dict(
            filter(lambda item: item[1] >= self.probability_cutoff, DICE_PROBABILITIES.items())
        )
        for roll, probability in filtered_rolls.items():
            dice = (math.floor(roll / 2), math.ceil(roll / 2))
            deterministic_action = Action(action.color, action.action_type, dice)
            game_copy = game.copy()
            game_copy.execute(deterministic_action, validate_action=False)
            action_node.children.append(StateNode(game_copy, probability))
        excess_probability = 1 - sum(filtered_rolls.values())
        if excess_probability > 0 and len(action_node.children) > 0:
            for state_node in action_node.children:
                state_node.probability += excess_probability / len(action_node.children)
    elif action.action_type == ActionType.BUY_DEVELOPMENT_CARD:
        ...
```

Kôd 3.15 *Isječak funkcije za ekspandiranje nedeterminističkih akcija (slučaj za bacanje kockica)*

S obzirom na to da Tree Search Player svoje ponašanje temelji na istim parametrima heurističke funkcije Value Function Playera, i na njega je moguće primijeniti genetski algoritam za optimizaciju parametara. Ipak, uz male korekcije protivničkih parametara koji su završili s pozitivnim vrijednostima, Tree Search Player izuzetno dobro funkcionira koristeći već naučene parametre od Value Function Playera. Pozitivne vrijednosti protivničkih parametara inače predstavljaju očiti nedostatak prilikom evaluacije MIN čvorova algoritmom minimaks.

3.5. Dorade radnog okvira

Pored implementacija novih igrača, u sklopu `catan_ai` paketa dodano je još nekoliko dorada. Napisane su nove, jednostavne i fleksibilne skripte za pokretanje velikih skupova (engl. *batch*) igri s prikupljanjem statistike, te za pokretanje igre kroz grafičko sučelje. Aliasi za navedene skripte su `play-batch` i `play-ui`. Za igranje kroz grafičko sučelje, prethodno moraju biti pokrenute sve komponente web aplikacije, ili svi Docker Compose servisi. Nove skripte podržavaju različite parametre, a upute za njihovo korištenje dostupne su uz argument `--help`.

Također, prilikom brojnih testiranja igrača i postupaka učenja, uočena je prilika za uvođenje paralelnog višeprocesorskog izvođenja (engl. *multiprocessing*) skupova igri. Korištenjem standardne biblioteke `concurrent`, implementirano je izvođenje igri u zasebnim procesima u „bazenu“ (engl. *pool*) s brojem radnika (engl. *worker*) jednakim broju dostupnih logičkih procesora umanjenom za 2 (`os.cpu_count() - 2`) (vidi Kôd 3.16).

```python
def play_batch_core(num_games, players, game_config, accumulators=[]):
    for accumulator in accumulators:
        if isinstance(accumulator, SimulationAccumulator):
            accumulator.before_all()

    with concurrent.futures.ProcessPoolExecutor(max_workers=max(os.cpu_count() - 2, 1)) as executor:
        futures = [executor.submit(play_game, players, game_config, deepcopy(accumulators)) for _ in range(num_games)]

        for future in concurrent.futures.as_completed(futures):
            game, accumulator_copies = future.result()
            for accumulator, copy in zip(accumulators, accumulator_copies):
                accumulator.join(copy)
            yield game

    for accumulator in accumulators:
        if isinstance(accumulator, SimulationAccumulator):
            accumulator.after_all()
```

Kôd 3.16 *Implementacija višeprocesorskog izvođenja skupova igri*

Konačno, kôd za treniranje genetskim algoritmom je značajno generaliziran, te se uz izradu jednostavnih funkcija po uzoru na oblikovni obrazac tvornica (engl. *factory*) može primijeniti na bilo kojeg igrača koji raspolaže nizom realnih ili cjelobrojnih parametara (vidi Kôd 3.17).

```python
def generate_vfp_params(noise: int) -> tuple[int, ...]:
    param_filepath = f'{PARAMETERS_DIR}{args.parameter_filename}' if args.parameter_filename is not None else None
    player = ValueFunctionPlayer(Color.RED, PlayerType.AGGRESSIVE, param_filepath)
    parameters = list(player.get_parameters())

    if noise > 0:
        for i in range(len(parameters)):
            parameters[i] += int(random.gauss(0, noise))

    return tuple(parameters)

def create_vf_player(parameters: tuple[int, ...], color: Color) -> ValueFunctionPlayer:
    player = ValueFunctionPlayer(color)
    player.set_parameters(parameters)
    return player
```

Kôd 3.17 *Funkcije za generiranje parametara i rekreiranje igrača u genetskom algoritmu (slučaj Value Function Playera)*

4. Rezultati

Vrijednosti parametara Value Function Playera naučene su kroz 200 iteracija genetskog algoritma nad populacijom od 60 jedinki, s 20 turnira po iteraciji i 20 odigranih partija po turniru (vidi Kôd 3.10 i Kôd 3.11). Standardna devijacija parametara iz inicijalne populacije iznosi 100, a za potrebe mutacije iznosi 50, uz 30% vjerojatnosti za mutaciju pojedinog parametra. U opisanoj konfiguraciji, provođenje genetskog algoritma traje otprilike 4 minute i 30 sekundi po iteraciji, uz 14 procesa-radnika na 16 logičkih jezgri. S obzirom na drastičan porast trajanja igri korištenjem Tree Search Playera, za njegove optimizirane parametre korištena je korigirana verzija parametara naučenih za Value Function Playera. Parametri protivničkih vrijednosti postavljeni su na vrijednost -10 u slučaju kada su učenjem postali pozitivni, jer ti parametri imaju puno veći utjecaj kod evaluacije MIN čvorova u Tree Search Playeru. Optimizirani parametri Neural Network Playera dobiveni su provođenjem genetskog algoritma nad pred-treniranim težinama mreže sa skrivenim slojevima veličina 32 i 16. Pred-treniranje je provedeno na skupu podataka od 500 tisuća nasumičnih uzoraka heurističke funkcije Value Function Playera s početnim vrijednostima parametara. Genetski algoritam također je proveden kroz 200 iteracija nad populacijom od 60 jedinki, s 20 turnira po iteraciji i 20 igri po turniru, ali uz inicijalnu standardnu devijaciju težina 2, odnosno 1 prilikom mutacije, uz 30% vjerojatnosti. Njegovo vrijeme izvođenja za Neural Network Playera iznosi otprilike 5 minuta po iteraciji, uz iste mogućnosti paralelizacije.

U tablici (Tablica 4.1) dostupan je pregled rezultata međusobnog testiranja novih agenata. Agenti su poredani počevši od najuspješnijeg, i za svakoga je prikazan postotak pobjeda nad sljedećim niže plasiranim agentom u uzorcima od 1000 odigranih igri jedan na jedan.

Tablica 4.1 *Rezultati testiranja agenata*

Napomena: + označava verziju igrača s optimiziranim parametrima

Agent	**Pobjede – 1v1 [%]**	**Broj igri**
Tree Search Player + (d=3)	60% vs Tree Search Player (d=3)	1000
Tree Search Player (d=3)	60% vs Value Function Player +	1000
Value Function Player +	70% vs Neural Network Player +	1000
Neural Network Player +	52% vs Value Function Player	1000
Value Function Player	99% vs Greedy VP Player	1000
Greedy VP Player	60% vs Weighted Random Player	1000
Weighted Random Player	55% vs Random Player	1000
Random Player	-	-
Neural Network Player	-	-

Može se reći da je rang igrača očekivan, no bitno je istaknuti skok u sposobnostima između posljednjeg *baseline* igrača – Greedy VP Playera, i prvog naprednog igrača – Value Function Playera, koji je djelomično i nadmašio očekivanja s 99% pobjeda u korist VFP-a, bez optimiziranih parametara. Najbolji agent, Tree Search Player, kroz testiranja se pokazao najefikasnijim kada pretražuje do dubine za jedan broj veće od broja igrača u igri. Daljnje povećavanje dubine ne donosi dovoljan napredak da opravda eksponencijalni porast u trajanju igre.

U tablici (Tablica 4.2) prikazani su rezultati testiranja najbolja četiri agenta u zajedničkom sukobu od 1000 odigranih igri. Ti agenti su: Tree Search Player s optimiziranim parametrima uz dubinu pretraživanja 3, Tree Search Player s početnim parametrima uz dubinu pretraživanja 3, Value Function Player s optimiziranim parametrima, te Neural Network Player s optimiziranim parametrima.

Tablica 4.2 *Rezultati sukoba između četiri najbolja agenta*

Napomena: + označava verziju igrača s optimiziranim parametrima

Agent	**Pobjede – 4 igrača [%]**	**Broj igri**
Tree Search Player + (d=3)	42%	1000
Tree Search Player (d=3)	22%	
Value Function Player +	26%	
Neural Network Player +	10%	

Ovi rezultati djelomično potvrđuju poredak dobiven testiranjem igrača jedan na jedan, osim zamjene pozicija neoptimiziranog Tree Search Playera i optimiziranog Value Function Playera. Porast sposobnosti optimiziranih agenata u igrama s većim brojem igrača zamijećen je i prilikom usporedbe istih s postojećim igračima radnog okvira Catanatron, te se može objasniti kao posljedica provođenja genetskog algoritma isključivo nad igrama od 4 igrača.

Neki zanimljiviji rezultati usporedbe novih agenata s postojećim igračima Catanatrona nalaze se u tablicama (Tablica 4.3 i Tablica 4.4). Za testiranje odabrani su prvi, drugi i četvrti najbolji igrači iz radnog okvira Catanatron, a to su redom Alpha-Beta Player, Value Function Player i Monte Carlo Tree Search Player. Treći najbolji igrač dostupan u Catanatronu je Greedy Playouts Player, no, s obzirom na trajanje njegovog odabira poteza (oko 3 minute za inicijalnu izgradnju naselja), nije upotrijebljen za usporedbu rezultata. Alpha-Beta Player i Value Function Player koriste slične principe donošenja odluke kao novoimplementirani Tree Search Player i Value Function Player, a MCTS Player implementira algoritam pretraživanja stabla Monte Carlo, te je prilikom testiranja suprotstavljen s trećim najboljim igračem ovog projekta – Neural Network Playerom.

Tablica 4.3 *Rezultati testiranja novih agenata protiv postojećih igrača radnog okvira Catanatron*
Napomena: 2v2 označava igre između 2 agenta jednog tipa i 2 agenta drugog tipa, sa zbrojenim iznosom pobjeda; **+** označava verziju igrača s optimiziranim parametrima; označava agente iz ovog projekta, a agente iz radnog okvira Catanatron

Agent	Pobjede – 1v1 [%]	Pobjede – 2v2 [%]	Broj igri
Tree Search Player + (d=3)	48%	62%	200
Alpha-Beta Player	52%	38%	
Value Function Player +	45%	51%	1000
Catanatron VFP	55%	49%	
Neural Network Player +	100%	100%	1000
MCTS Player	0% (< 5 igri)	0% (< 5 igri)	

Tablica 4.4 *Rezultati sukoba Tree Search Playera s najboljim agentima radnog okvira Catanatron*
Napomena: + označava verziju igrača s optimiziranim parametrima; označava agente iz ovog projekta, a agente iz radnog okvira Catanatron

Agent	Pobjede – 4 igrača [%]	Broj igri
Tree Search Player + (d=3)	46%	200
Alpha-Beta Player	33%	
Catanatron VFP	21%	
MCTS Player	0%	

Rezultati ovog testiranja jasno pokazuju da novi agenti imaju prednost nad postojećima u domeni za koju su i trenirani – igre s 4 igrača. S obzirom na to, odabrani proces učenja, a i same implementacije igrača, mogu se smatrati vrlo uspješnima.

Neural Network Player jedini je napredni igrač koji ponekad donosi objektivno pogrešne odluke, poput gradnje početnog naselja na obali, i to ne u luci, time smanjujući

vlastitu proizvodnju resursa. Ovo se može obrazložiti činjenicom da je njegovo početno stanje težina ipak potpuno nasumično. Igranjem kroz grafičko sučelje protiv heurističkih agenata, može se vidjeti logika iza njihovih poteza i uglavnom se ne događa da odigraju potez koji nema smisla u trenutnoj situaciji. S druge strane, nijedan od igrača nije pretjerano sposoban voditi dugotrajnu strategiju. Tree Search Player, koji bi mogao biti sposoban za to, testiran je na relativno malim ograničenjima dubine – 4 ili manje, gdje dugotrajne strategije nisu ostvarive. S obzirom na to da igrač može odigrati više akcija tijekom svog poteza, i da svaki sljedeći igrač mora početi bacanjem kockica, ovakve dubine su obično samo dovoljne da agent može isplanirati nekoliko povezanih akcija u svom trenutnom potezu. Ipak, to se ne pokazuje kao jako veliki nedostatak, jer su neki aspekti dugotrajnih strategija, poput vrednovanja slobodnih lokacija za izgradnju, štednje za naselja i gradove, ali i opreza s brojem karata u ruci, uspješno implementirani samom heurističkom funkcijom. Sposobnost koordinacije više akcija u potezu daje taman dovoljnu prednost da osnovni Tree Search Player nadjača optimiziranog Value Function Playera.

Sve u svemu, primjećuje se da se napredni agenti ne fokusiraju na samo jednu strategiju, već su optimizacijom parametara razvili zdrav balans između skupljanja bodova izgradnjom naselja i gradova, dužinom ceste, razvojnim kartama, te vojskom. Suprotno tome, postojeći igrači radnog okvira Catanatron znatno se više oslanjaju na povećanje broja naselja, a dužinom ceste i korištenjem razvojnih karata gotovo niti ne konkuriraju novim agentima (vidi Slika 4.1). Sve implementirane igrače moguće je pobijediti uz malo truda, no nije uvijek lako. Uostalom, teško je dati realnu procjenu njihovih sposobnosti kada nedostatak domaće trgovine u trenutnoj verziji Catanatrona predstavlja hendikep agentima, kao i ljudskim igračima.

	WINS	AVG VP	AVG SETTLES	AVG CITIES	AVG ROAD	AVG ARMY	AVG DEV VP
Tree Search Player	91	8.11	2.16	1.34	0.57	0.53	1.06
Alpha-Beta Player	66	7.21	2.71	1.93	0.30	0.01	0.04
Catanatron VFP	43	6.60	2.73	1.75	0.12	0.01	0.12
MCTS Player	0	2.40	2.00	0.06	0.00	0.01	0.27

Slika 4.1 *Sažetak sukoba Tree Search Playera s najboljim agentima radnog okvira Catanatron*

5. Moguća poboljšanja

U trenutnom skupu ulaza heurističke funkcije Value Function Playera postoji nekoliko vrijednosti koje nisu potpuno nezavisne. Primarno je riječ o broju pobjedničkih bodova, gradova i naselja. Kako bi se izbjeglo prisustvo kolinearnosti u modelima koji koriste ove ulaze, bilo bi korisno pronaći alternativnu kombinaciju ulaza kojom pobjednički bodovi ne moraju biti dodatno definirani, već su jednoznačno određeni kombinacijom drugih parametara. Ovo bi moglo posebno pomoći prilikom pred-treniranja Neural Network Playera. Nadalje, odabir akcije u Value Function Playeru i Neural Network Playeru bi se mogao podijeliti ovisno o vrsti akcije. Tako se, za specifične situacije poput pomicanja razbojnika, ne bi morao uzimati u obzir velik broj ulaza koji će definitivno ostati nepromijenjeni neovisno o odigranoj akciji. Prilikom testiranja također je primijećeno da faza inicijalne gradnje uzrokuje jedno od najvećih grananja stabla igre kod Tree Search Playera, te time velikim dijelom doprinosi dužem trajanju cijele igre. Izdvajanje procesa inicijalne gradnje iz algoritma minimaks moglo bi dovesti do primjetnog ubrzanja igrača. Na sličan način bi se mogao izdvojiti proces odabira nove lokacije za razbojnika. S obzirom na to da je riječ o nedeterminističkoj akciji zbog krađe protivnikovog resursa, moglo bi biti korisno unaprijed odabrati optimalnu poziciju za razbojnika, te samo kroz odabranu akciju dalje granati stablo.

Osim optimizacija, treba proučiti mogućnost dodavanja novih ulaza u heurističku funkciju. Primjeri vrijednosti koje bi mogle značajno utjecati na igru su prosječna udaljenost među naseljima, te broj odvojenih nizova cesti. Tim podacima bi se moglo predočiti je li igrač previše rasprostranjen, što potencijalno omogućuje drugim igračima da ga opkole i blokiraju.

Bilo bi dobro isprobati i alternativne pristupe učenju. Metode podržanog učenja, poput Q-učenja (engl. *Q-learning*), mogu dati dodatnu slobodu agentima u pronalasku boljih strategija kombiniranjem istraživanja i eksploatacije (engl. *exploration and exploitation*). Catanatron već sadrži modul namijenjen radu s podržanim učenjem preko biblioteke Gym[22], no njegova implementacija nije detaljno analizirana ovim radom. Postoji i

[22] https://www.gymlibrary.ml/

mogućnost dorade trenutnog procesa učenja kombiniranjem različitih brojeva igrača u igrama unutar genetskog algoritma. Bolji rezultati heurističkih agenata u igrama s 4 igrača ukazuju na to da bi se na ovaj način mogao ostvariti dodatan napredak i u preostalim konfiguracijama igri.

Zaključak

U ovom radu detaljno je opisan pristup rješavanju problema izrade agenta za društvenu igru Catan. Rezultat rada su tri polazna (*baseline*) igrača, korisni kao početna točka za analizu uspjeha, te tri napredna igrača koji primjenjuju različite metode odabira poteza na temelju trenutnog stanja igre i usađenog znanja o raznim konceptima Catana. Uspjesi koje postižu implementirani igrači su zadovoljavajući, no postoji još mnogo mjesta za napredak. Današnje stanje AI tehnologije i strojnog učenja već nudi mnoge druge načine za rješavanje ovakvih problema, i bilo bi zanimljivo vidjeti sva njihova postignuća. Neke ideje za buduće pokušaje savladavanja ove ili sličnih igri već su iznesene u sklopu ovog rada, a korišteni pristup može poslužiti kao inspiracija i dobar primjer za daljnja istraživanja i pronalazak novih rješenja.

Dokle god ima mjesta za napredak umjetne inteligencije, društvene igre i videoigre mogu biti neiscrpan izvor izazova za razvoj novih i boljih rješenja, kao i za njihovo testiranje i demonstraciju sposobnosti. Također, ovakvim projektima moguće je na zabavan način približiti područje umjetne inteligencije velikom broju ljudi koji pokreću masovnu globalnu industriju igara.

Ako je konačan cilj stvaranje sustava koji je sposoban učiti, generalizirati, donositi odluke i zaključke, te rješavati različite kompleksne probleme, onda se agenta koji može savladati neke od najkompetitivnijih strateških igri na svijetu zaista može nazvati inteligentnim.

Literatura

[1] *About Us*, CATAN. Poveznica: https://www.catan.com/about-us; pristupljeno 22. lipnja 2022.

[2] *CATAN*, CATAN. Poveznica: https://www.catan.com/catan; pristupljeno 22. lipnja 2022.

[3] Cabrera, B. *Settlers of Catan*, GitHub Pages. Poveznica: https://bryantcabrera.github.io/Settlers-of-Catan/; pristupljeno 22. lipnja 2022.

[4] Teuber, K. *Game Rules & Almanac*, CATAN. Poveznica: https://www.catan.com/sites/prod/files/2021-06/catan_base_rules_2020_200707.pdf; pristupljeno 22. lipnja 2022.

[5] *Settlers of Catan Expansions | Official Rules*, Ultra BoardGames. Poveznica: https://www.ultraboardgames.com/catan/expansions.php; pristupljeno 22. lipnja 2022.

[6] Monin, J.D. *Java Settlers of Catan*, nand.net. Poveznica: https://nand.net/jsettlers/; pristupljeno 22. lipnja 2022.

[7] Monin, J.D. *JSettlers2*, GitHub. Poveznica: https://github.com/jdmonin/JSettlers2; pristupljeno 22. lipnja 2022.

[8] Vombatkere, K. *CatanAI*, GitHub. Poveznica: https://github.com/kvombatkere/Catan-AI; pristupljeno 22. lipnja 2022.

[9] Collazo, B. *5 Ways NOT to Build a Catan AI*, Medium, (2021, kolovoz). Poveznica: https://medium.com/@bcollazo2010/5-ways-not-to-build-a-catan-ai-e01bc491af17; pristupljeno 22. lipnja 2022.

[10] Collazo, B. *Catanatron*, GitHub. Poveznica: https://github.com/bcollazo/catanatron; pristupljeno 22. lipnja 2022.

[11] Čupić, M., Dalbelo Bašić, B., Šnajder, J., nastavni materijal kolegija Umjetna inteligencija, Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu. Poveznica: https://www.fer.unizg.hr/predmet/umjint/materijali; pristupljeno 22. lipnja 2022.

[12] Lin, Y. *Game Trees*, Open Computing Facility, Berkley, University of California. Poveznica: https://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html; pristupljeno 22. lipnja 2022.

[13] Abdulazeez, S., Nasereddin, H. H. O. *Enhanced Solutions for Misuse Network Intrusion Detection System using SGA and SSGA*, ResearchGate, (2015, lipanj). Poveznica: https://www.researchgate.net/publication/290789986_Enhanced_Solutions_for_Misuse_Network_Intrusion_Detection_System_using_SGA_and_SSGA; pristupljeno 22. lipnja 2022.

[14] Horton, P., Jaboyedoff, M., Obled, C. *Global Optimization of an Analog Method by Means of Genetic Algorithms*, ResearchGate, (2017, travanj). Poveznica: https://www.researchgate.net/publication/315384001_Global_Optimization_of_an_Analog_Method_by_Means_of_Genetic_Algorithms; pristupljeno 22. lipnja 2022.

[15] Kim, R. *Debate on the Relationship between Neural Network and the Brain*, Web Publishing, NYU, (2020, ožujak). Poveznica: https://wp.nyu.edu/yungjurick/2020/03/15/debate-on-the-relationship-between-neural-network-and-the-brain/; pristupljeno 22. lipnja 2022.

[16] Esfe, M. H., Eftekhari, S. A., Hekmatifar, M., Toghraie, D. *A well-trained artificial neural network for predicting the rheological behavior of MWCNT–Al2O3 (30–70%)/oil SAE40 hybrid nanofluid*, Nature, (2021, kolovoz). Poveznica: https://www.nature.com/articles/s41598-021-96808-4; pristupljeno 22. lipnja 2022.

[17] *Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)*, Geeks for Geeks, (2021, kolovoz). Poveznica: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/; pristupljeno 22. lipnja 2022.

Sažetak

Razvoj agenta za društvenu igru Catan

Ovaj rad opisuje razvoj šest računalnih igrača, odnosno agenata za stratešku društvenu igru Catan. Pravila i koncepti igre detaljno su opisani, kako bi se lakše obrazložile ideje iza pojedinih igrača. Za potrebe razvoja i igranja na računalu, isprobana su tri radna okvira za Catan, i analizirani su u sklopu ovog rada. Agenti su razvijeni oko različitih metoda za odabir poteza, poput pretraživanja prostora stanja i neuronskih mreža. Kod nekih se koriste i tehnike strojnog učenja za optimizaciju parametara. Njihove sposobnosti uspoređene su na velikim uzorcima odigranih igri, a njihovo ponašanje je evaluirano igranjem protiv njih kroz grafičko sučelje radnog okvira Catanatron. Catanatron implementira nekoliko postojećih igrača, te su novi agenti uspoređeni i s njima. Na kraju rada, iznesene su i razne ideje za budući razvoj te napredak projekta.

Ključne riječi: Catan, društvena igra, umjetna inteligencija, agent, strategija, strojno učenje, heuristika, pretraživanje prostora stanja, neuronska mreža, genetski algoritam, neuroevolucija

Summary

Development of an agent for the Catan boardgame

This paper describes the development of six computer players, i.e., agents for the Catan strategy board game. The rules and concepts of the game are described in detail, to help explain the ideas behind individual players. For the purposes of development and testing, three Catan frameworks have been tested, and are analyzed as part of this paper. Agents have been developed around various methods for selecting their actions, such as state space search and neural networks. With some of them, machine learning techniques have been used to optimize their parameters. Their capabilities have been compared on large samples of games played, and their behavior has been evaluated by playing against them through Catanatron framework's graphical user interface (GUI). Catanatron implements several existing players, and the new agents have been compared to those as well. At the end of the paper, various ideas for future development and improvements of the project are presented.

Keywords: Catan, boardgame, artificial intelligence, agent, strategy, machine learning, heuristics, state space search, neural network, genetic algorithm, neuroevolution

Privitak

Instalacija programskog rješenja

Programsko rješenje nalazi se u javnom GitLab[23] repozitoriju na poveznici https://gitlab.com/iskoric/catanatron. Nakon kloniranja repozitorija, potrebno je slijediti upute u datoteci `README.md` modula `catan_ai`.

[23] https://gitlab.com/