Implementacija MCTS algoritma u obrazovnoj aplikaciji za igranje društvene igre "4 u nizu"

Pipalović, Denis

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 621

IMPLEMENTACIJA MCTS ALGORITMA U OBRAZOVNOJ APLIKACIJI ZA IGRANJE DRUŠTVENE IGRE "4 U NIZU"

Denis Pipalović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 621

IMPLEMENTACIJA MCTS ALGORITMA U OBRAZOVNOJ APLIKACIJI ZA IGRANJE DRUŠTVENE IGRE "4 U NIZU"

Denis Pipalović

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 621

Pristupnik:	**Denis Pipalović (0036525518)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	doc. dr. sc. Tomislav Jagušt

Zadatak: **Implementacija MCTS algoritma u obrazovnoj aplikaciji za igranje društvene igre "4 u nizu"**

Opis zadatka:

Monte Carlo Tree Search (MCTS) algoritam predstavlja jednu od najpopularnijih metoda donošenja odluka u računalnim simulacijama društvenih igara. Sustavi zasnovani na ovom algoritmu pokazali su se uspješni u učenju i igranju niza društvenih, kartaških i računalnih igara, pogotovo u kombinaciji s neuronskim mrežama ili npr. Minimax algoritmom. U sklopu ovog diplomskog rada potrebno je istražiti različite strategije igranja igre "4 u nizu"(engl. "Connect 4"), proučiti literaturu o naprednim AI tehnikama za ovu vrstu igara, te razviti aplikaciju koja omogućava igranje igre protiv računalnog igrača koji za određivanje poteza koristi MCTS. Aplikacija treba korisniku nuditi preporuke za poteze (i objašnjenja danih preporuka) te opciju prilagodbe algoritma u svrhu promjene težine igre. Potrebno je istražiti i implementirati nekoliko varijanti algoritma (npr. kombinacija MCTS i Minimax) te ih međusobno usporediti.

Rok za predaju rada: 28. lipnja 2024.

SADRŽAJ

1. Uvod

Monte Carlo Tree Search (MCTS) algoritam jedna je od najpopularnijih metoda donošenja odluka u simulacijama društvenih igara. Sustavi zasnovani na MCTS algoritmu i njegovim varijacijama pokazali su se vrlo uspješnima u igranju raznih tipova igara. U sklopu ovog diplomskog rada istražene su neke varijante ovog algoritma.

Također, razvijena je i aplikacija za igranje i učenje igranja društvene igre „4 u nizu" (engl. *„Connect 4"*). Aplikacija za učenje igranja društvene igre *„Connect 4"* predstavlja inovativan pristup kombiniranju zabave i učenja te razvijanja strategije i zaključivanja.

Cilj aplikacije je omogućiti korisniku natjecanje protiv računalnog protivnika. Računalni protivnik koristi MCTS za određivanje svojih poteza te se ovaj algoritam također koristi i kako bi se procijenio svaki mogući potez igrača i pružila povratna informacija o njegovoj kvaliteti. Ovo omogućuje bolje razumijevanje strategije i logiku iza svakog poteza, što u konačnici igraču pomaže poboljšati njegove poteze.

Jedna od ključnih značajki aplikacije je mogućnost pregledavanja starih igara i poteza. Korisnici mogu analizirati svoje prethodne igre, prepoznati greške i razviti strategije za buduće igre. Unutar same igre koju pregledavaju, korisnici imaju mogućnost djelovati i odabrati drugi potez, što im omogućuje isprobavanje različitih strategija i učenje iz svojih pogrešaka. Aplikacija korisniku nudi i prilagodbu težine igranja, odnosno kvalitete poteza računalnog protivnika.

2. Društvene igre

Društvene igre (engl. *board games*) jedan su od najčešćih oblika zabave među populacijom koja nije zaokupljena digitalizacijom, pametnim mobitelima i računalima; ukratko svime što smanjuje socijalizaciju među ljudima. Većina društvenih igara igra se na nekoj površini kao što je stol, čime je povećana međusobna interakcija. Dok su neke društvene igre ograničene na samo dva igrača, druge pak igre omogućavaju zabavu za cijelu obitelj ili grupu prijatelja. Društvene igre mogu se podijeliti u veliki broj skupina, ovisno o kompleksnosti, načinu igranja, broju igrača, svrsi i slično. Neke od osnovnih skupina društvenih igara su:

- strateške igre - igre koje zahtijevaju planiranje i korištenje raznih strategija, npr. „šah" i „4 u nizu";
- kartaške igre - igre koje se primarno igraju kartama, npr. „belot" i „Uno";
- obrazovne igre - igre koje za cilj imaju učenje, npr. „Trivial pursuit";
- logičke igre - igre u kojima igrači pokušavaju otkriti i povezati podatke kako bi dobili informaciju od značaja, npr. „Secret Hitler" i „Werewolf";
- obiteljske igre - igre koje su dizajnirane tako da budu zanimljive za cijelu obitelj, odnosno sve uzraste, npr. „Labyrinth" i „Monopoly".
- ...

Ove skupine igara su samo jedne od nekih te se međusobno ne isključuju. Svaku igru moguće je opisati s više skupina te se najčešće ne može reći da neka igra pripada isključivo u jednu od njih.

Društvene igre su dobar način zabave, poticatelj međusobne komunikacije i razvoja vještina kao što su strategija, timski rad i logika.

2.1. Korisnost društvenih igara

Društvene igre nisu samo oblik zabave već kao takve mogu biti i korisne. Kao što navodi Christy Edwards[10], društvene igre mogu biti korisne za razvoj raznih vještina,

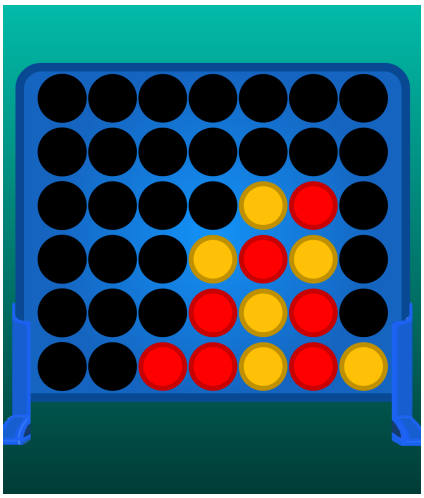
od socijalnih do kognitivnih. Također, društvene igre potiču međusobnu komunikaciju, socijalne vještine i povezivanje s drugim ljudima. Edwards navodi i rezultate južnokorejskog istraživanja iz 2017. godine, koje je pokazalo da igranje društvenih igara može smanjiti stres te povećati smirenost. Kako društvene igre navode na smijeh tako se potiče izlučivanje endorfina, jednog od hormona sreće. Osim endorfina, igranje društvenih igara potiče i izlučivanje dopamina te serotonina, što ublažava simptome anksioznosti. Izlučivanje navedenih hormona povećava entuzijazam te može pozitivno utjecati na mentalno zdravlje. U francuskom istraživanju[2] iz 2013. godine, koje je uključivalo 3675 ispitanika, pokazalo se da redovno igranje društvenih igara može smanjiti rizik od Alzheimerove bolesti za 15%.

S obzirom na sve istaknuto u ovom potpoglavlju može se zaključiti da društvene igre mogu biti korisne za razvoj kognitivnih vještina te poticanje razmišljanja, logike, strategije, planiranja, bolje memorije, sposobnosti donošenja odluka i kritičkog razmišljanja.

2.2. Connect 4

„4 u nizu" (engl. *„Connect 4"*) poznata je društvena igra koja je prvi put objavljena prije točno 50 godina, u veljači 1974. godine. Ova igra tip je *„m,n,k igre"*, gdje *m* i *n* predstavljaju dimenziju ploče $m \times n$, dok *k* predstavlja broj uzastopnih tokena jednog igrača potrebnih za pobjedu. Uzastopne tokene potrebne za pobjedu, igrač osim u jednom retku ili jednom stupcu, može postaviti i u dijagonalnom smjeru. Najpopularnija verzija *„m,n,k igre"* igre je *križić-kružić* (engl. *tic-tac-toe*), za koju vrijedi m = n = k = 3. Tako za *„4 u nizu"* vrijedi m = 7, n = 6, k = 4. Igrača ploča (slika 2.1) sastoji se od 7 stupaca, u svakom po 6 redaka. Za ostvariti pobjedu igrač treba postaviti svoje tokene, kao što ime igre kaže, *„4 u nizu"*.

Uobičajene boje tokena u ovoj igri su crvena i žuta. Tokeni se na ploču umeću na najniže slobodno mjesto u nekom stupcu, a igrači poteze rade naizmjenično. Najčešći oblik ove igre napravljen je tako da ploča stoji uspravno, dok se tokeni umeću odozgo, stoga sami padaju na najniže slobodno mjesto u stupcu.



Slika 2.1: Igrača ploča igre „4 u nizu”

3. Pretraga prostora stanja

Pretraživanje prostora stanja (engl. *state-space search*) jedan je od temeljnih koncepata umjetne inteligencije. U pretrazi prostora stanja, problem se definira kao skup početnih stanja, skup mogućih akcija, odnosno prijelaza i skup ciljnih stanja. Cilj pretraživanja prostora stanja je pronaći niz akcija koji vodi od početnog do ciljnog stanja. Proces pretraživanja prostora stanja može se izvoditi raznim algoritmima od kojih neki koriste heuristiku, a neki ne. Korištenje heuristike može značajno ubrzati pretragu prostora stanja i dolaska do rješenja. No, ovisno o heuristici, može se dogoditi da algoritam ne pronađe optimalno rješenje. Osim suboptimalnosti, najčešće je vrlo teško definirati heuristiku za konkretni problem. Problem može biti definiran tako da prijelazi imaju jednake ili različite cijene. U nastavku su ukratko opisani neki od najpoznatijih algoritama za pretragu prostora stanja.

3.1. Algoritmi za pretragu prostora stanja

3.1.1. DFS

Pretraživanje u dubinu (engl. *Depth-First Search*, DFS) spada među najjednostavnije algoritme za pretragu prostora stanja za probleme s jednakom cijenom prijelaza. Ovaj algoritam pretražuje prostor stanja u dubinu, tj. prvo pretražuje sve moguće akcije iz jednog čvora, zatim pretražuje akcije za sljedeći čvor. S obzirom na opisani način pretraživanja, DFS algoritam često pronalazi rješenje koje nije optimalno. Njegova prednost, osim same jednostavnosti za implementaciju, je što zahtjeva manje memorije od nekih drugih algoritama, primjerice algoritma BFS 3.1.2. Ovaj pristup pretraživanja prostora stanja često se koristi u problemima gdje je potrebno pronaći bilo koje rješenje, a ne nužno ono optimalno.

3.1.2. BFS

Pretraživanje u širinu (engl. *Breadth-First Search*, BFS) je algoritam koji pretražuje prostor stanja u širinu s ciljem pronalaska optimalnog rješenja. Postupak pretraživanja prostora stanja u širinu sličan je pretraživanju u dubinu. Umjesto pretraživanja svih mogućih čvorova iz jednog čvora, prvo se pretražuju svi čvorovi koji su udaljeni za jedan korak od početnog čvora. Zatim svi čvorovi koji su udaljeni za dva koraka i tako dalje. Kako bi to bilo moguće, potrebno je koristiti značajno više memorije nego kod DFS algoritma. Prednost toga je da će algoritam pronaći optimalno rješenje. S obzirom na potrebu za većom količinom memorije, ovaj algoritam teško se može koristiti u problemima gdje je prostor stanja velik. Ovaj pristup pretraživanju prostora stanja često se koristi u problemima gdje je potrebno pronaći najkraći put do rješenja, uz uvjet da sve akcije imaju jednaku cijenu.

3.1.3. A*

Pretraživanje A* (engl. *A* Search*) je jedan od najpoznatijih algoritama za pretraživanje prostora stanja s različitim cijenama prijelaza. Algoritam A* koristi cijenu puta iz početnog čvora te heuristiku za taj čvor kako bi odredio koji čvor pretražiti sljedeći. Prvo pretražuje čvorove koji imaju manju sumu ove dvije vrijednosti. Heuristika najčešće predstavlja predviđenu cijenu puta od trenutnog do ciljnog stanja. Algoritam A* je optimalan, tj. uvijek pronalazi optimalno rješenje, ali pod uvjetom da je heuristika optimistična ili dopustiva (engl. *optimistic or admissible*). Optimističnost heuristike znači da heuristika nikada (niti za jedan čvor) ne smije precijeniti cijenu puta do ciljnog stanja, odnosno predviđena cijena puta uvijek je manja ili jednaka stvarnoj cijeni puta. Ovaj pristup pretraživanju prostora stanja često se koristi u problemima gdje je potrebno pronaći najkraći put do rješenja, uz uvjet da akcije imaju različite cijene.

3.1.4. Minimax

Algoritam minimax primjenjuje se u teoriji igara i umjetnoj inteligenciji, posebno u kontekstu igara s nultom sumom, u kojoj postiže optimalne poteze. Igre s nultom sumom su tip igara u kojima igrači imaju strogo različite interese, što znači da dobitak za jednog igrača predstavlja gubitak za drugog.

Minimax algoritam djeluje na principu pretrage prostora stanja kroz stablo mogućih poteza. Glavna ideja je maksimizacija vlastitog dobitka i istovremeno minimizacija dobitka protivnika. Svaka razina stabla predstavlja poteze jednog igrača, pri čemu se

naizmjenično maksimiziraju i minimiziraju vrijednosti čvorova stabla.

Na najdubljoj razini stabla, evaluira se rezultat igre te se taj rezultat vraća prema korijenu stabla, gdje se donosi odluka o potezu. Jedna od ključnih prednosti Minimax algoritma je što garantira pronalazak optimalnog poteza u igrama s potpunom informacijom. Međutim, problem za minimax algoritam su igre s velikim brojem mogućih poteza ili s djelomičnom informacijom, gdje je prostor stanja prevelik za potpunu pretragu. Tada se mogu koristiti različite tehnike za smanjenje prostora pretrage; jedna od njih je alfa-beta rez (engl. *alpha-beta pruning*).

Minimax algoritam je temelj za mnoge naprednije tehnike u teoriji igara, poput Monte Carlo Tree Search algoritma. Primjena ovog algoritma nije ograničena samo na područje igara, već se može koristiti u raznim područjima u kojima je potrebno donošenje odluka.

3.1.5. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) jedan je od najpopularnijih algoritama za pretragu prostora stanja u igrama. Ovaj algoritam koristi Monte Carlo metodu za pretraživanje prostora stanja, tj. simulira veliki broj igara kako bi se odredila kvaliteta pojedinog poteza. Umjesto nasumičnog simuliranja igara iz početnog čvora, MCTS algoritam koristi stablo pretrage koje se dinamički gradi tijekom pretrage, dok se simulacije igara izvode iz listova stabla.

S obzirom na to da se stablo pretrage gradi prema određenim pravilima, MCTS algoritam smanjuje prostor pretrage, odnosno fokusira se na najbolje poteze. Za razliku od Minimax ili BFS algoritma, MCTS ne pretražuje sve moguće poteze. Na ovaj se način MCTS algoritam može koristiti u igrama s velikim prostorom stanja, gdje je pretraga svih mogućih poteza nemoguća. Smanjenjem prostora pretrage, smanjuje se i potrebna količina memorije za pretragu, što je jedna od prednosti ovog algoritma.

S obzirom na značaj MCTS-a u ovom radu, ovaj algoritam je opširnije opisan u zasebnom poglavlju (4).

4. Monte Carlo Tree Search

U radu „Monte-Carlo Tree Search: A New Framework for Game AI", ističe se da je ključni faktor pri implementaciji AI za računalne igrice funkcija za procjenu kvalitete stanja igre [1]. Jasno je da kvaliteta stanja igre ovisi o mnogim faktorima te je teško ili nemoguće napisati jednu funkciju koja će vrijediti za sve igre. Osim toga, problematično je naći optimalnu funkciju za procjenu stanja točno određene igre. Od uvođenja Monte Carlo stabla pretraživanja 2006. godine, računalni programi za igru Go su značajno napredovali, prelazeći s niskog *kyu nivoa* na profesionalni *dan nivo* u Go igri na ploči 9×9, čime je MCTS stekao svoju popularnost [18].

MCTS je algoritam tipa *"najbolji prvi"* (engl. *best-first*), što znači da se u svakom koraku pretraživanja odabire čvor koji se čini najboljim te se pretraživanje nastavlja iz njega. Ukratko, MCTS može se opisati kao algoritam koji koristi Monte Carlo metodu za simulaciju igara, što znači da se igra simulira veliki broj puta kako bi se odredila kvaliteta poteza i donjela odluka o najboljem potezu. Uz Monte Carlo metodu, MCTS koristi i stablo pretraživanja kako bi se smanjio prostor pretrage te se fokusirao na najbolje poteze. Stablo pretraživanja se gradi iterativno, a svaka iteracija se sastoji od četiri koraka.

4.1. Koraci algoritma

Monte Carlo Tree Search u svakoj iteraciji koristi četiri ključna koraka: selekciju, ekspanziju, simulaciju i propagaciju unatrag. Svaki od tih koraka ima ulogu u procesu izgradnje stabla pretraživanja i omogućava algoritmu da efikasno istražuje veliki prostor stanja.

4.1.1. Selekcija

Svaka iteracija MCTS algoritma započinje korakom selekcije. Cilj ovog koraka je odabrati čvor koji će se proširiti u sljedećem koraku. Postupak selekcije počinje od

korijena te se kreće prema listovima stabla pretraživanja. Korak selekcije je iterativan, što znači da se u svakom njenom koraku odabire čvor koji se čini najboljim. Odabir najboljeg čvora može se vršiti na razne načine, no najčešće se koristi UCB1 (engl. *Upper Confidence Bound 1*) formula. UCB1 formulom dobiva se vrijednost temeljena na dvije komponente: prosječnoj vrijednosti čvora i komponenti istraživanja. Prosječna vrijednost čvora računa se kao omjer između ukupne nagrade i broja posjeta čvoru, dok se komponenta istraživanja računa kao korijen prirodnog logaritma broja posjeta čvora roditelja podijeljenog s brojem posjeta promatranog čvora. Odnosno formula za UCB1 izgleda ovako:

$$UCB1 = \frac{w_i}{n_i} + C \cdot \sqrt{\frac{\ln N}{n_i}} \tag{4.1}$$

gdje je:

- w_i - ukupna nagrada čvora
- n_i - broj posjeta čvoru
- C - konstanta koja određuje razinu istraživanja
- N - broj posjeta čvora roditelja

Kao najbolji čvor odabire se čvor koji ima najveću vrijednost dobivenu UCB1 formulom. Naime, komponenta istraživanja 4.2 pomaže algoritmu istražiti i manje posjećene čvorove, dok prosječna vrijednost čvora potiče nastavak istraživanja čvorova koji su se pokazali dobrima u prošlosti.

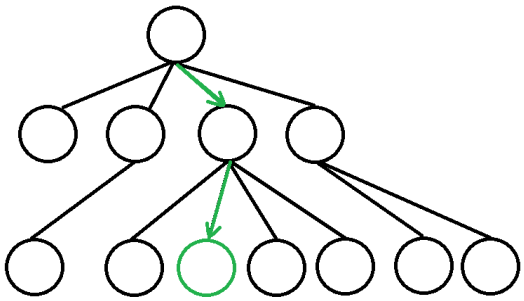
$$\sqrt{\frac{\ln N}{n_i}} \tag{4.2}$$

Komponenta istraživanja može se rastaviti na sljedeće dijelove:

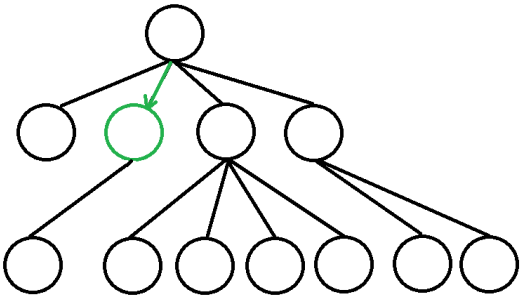
- $\ln N$ - prirodni logaritam broja posjeta čvora roditelja. Što je veći broj posjeta čvora roditelja, to je ovaj dio komponente istraživanja veći, a time i komponenta istraživanja. Logaritam se koristi kako bi se spriječilo preveliko istraživanje čvorova koji su već prilično istraženi;
- n_i - broj posjeta promatranog čvora. Što je veći broj posjeta promatranog čvora, to je ovaj dio komponente istraživanja veći, a time je komponenta istraživanja manja;
- $\ln N/n_i$ - omjer između prirodnog logaritma broja posjeta čvora roditelja i broja posjeta promatranog čvora. Kako se broj posjeta čvora roditelja povećava, a broj posjeta promatranog čvora ostaje isti, komponenta istraživanja se povećava;

– $\sqrt{\ldots}$ - korijen omjera. Korijen se koristi kako bi se komponenta istraživanja uravnotežila s prosječnom vrijednošću čvora.

Jednom kada se dođe do lista ili čvora koji nije potpuno proširen, selekcija se zaustavlja i prelazi se na korak ekspanzije. Slika 4.1 prikazuje postupak selekcije kada se iterativnim postupkom dosegne list stabla, dok slika 4.2 prikazuje postupak selekcije kada se dosegne čvor koji nije potpuno proširen.



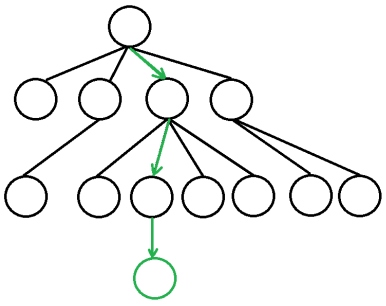
Slika 4.1: Postupak selekcije u MCTS algoritmu - list stabla



Slika 4.2: Postupak selekcije u MCTS algoritmu - nepotpuno proširen čvor

4.1.2. Ekspanzija

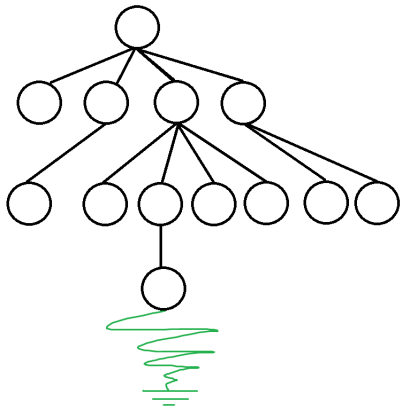
Nakon selekcije slijedi korak ekspanzije. Selekcijom je odabran čvor koji će se proširiti, a korakom ekspanzije se proširuje. Proširenje čvora postupak je dodavanja djeteta čvoru. Dijete čvora predstavlja jedan od mogućih poteza koji se može odigrati u trenutnom stanju igre. Dijete s kojim se proširuje čvor odabire se nasumično iz skupa mogućih poteza koji se mogu odigrati iz trenutnog stanja igre, ali koji nisu već dodani kao djeca čvoru. Slika 4.3 prikazuje postupak ekspanzije u MCTS algoritmu, gdje je novonastalo dijete označeno zelenom bojom. Nakon što se čvor proširi, prelazi se na korak simulacije.



Slika 4.3: Postupak ekspanzije u MCTS algoritmu

4.1.3. Simulacija

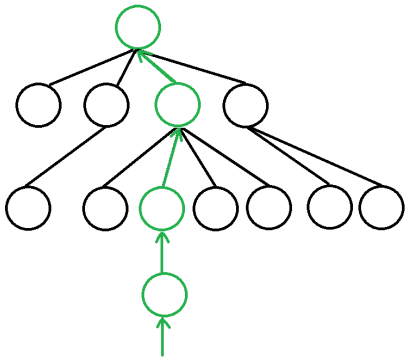
Korak simulacije, također se naziva i *rollout*, a predstavlja simulaciju od trenutnog stanja do kraja igre. U ovom koraku, igra se simulira do kraja, a potezi se odabiru prema *rollout* politici (engl. *rollout policy*). *Rollout* politika u osnovnoj verziji MCTS algoritma je nasumična, što znači da se potezi odabiru slučajno. Slika 4.4 prikazuje postupak simulacije u MCTS algoritmu. Zelenom bojom prikazana je simulacija igre od trenutnog stanja do kraja igre.



Slika 4.4: Postupak simulacije u MCTS algoritmu

4.1.4. Propagacija unatrag

Po završetku simulacije, slijedi korak propagacije unatrag (engl. *backpropagation*). Ovo je posljednji korak u svakoj iteraciji MCTS algoritma te ima za cilj ažuriranje statistika čvorova koji su posjećeni tijekom simulacije. Taj postupak ažuriranja statistika čvorova započinje od čvora koji je proširen u koraku ekspanzije te se kreće prema korijenu stabla. U svakom koraku propagacije unatrag, ažuriraju se statistike čvorova prema rezultatu simulacije. Ažuriranje statistika čvora obično se vrši inkrementiranjem broja posjeta čvoru te dodavanjem nagrade koja je dobivena tijekom simulacije. Nagrada koja se dodaje čvoru obično je rezultat igre. Igraču koji je pobijedio dodaje se pozitivna nagrada (obično +1), a igraču koji je izgubio negativna, odnosno oduzima se vrijednost pozitivne nagrade. Ako je rezultat igre neriješen niti jedan igrač ne dobiva nagradu. Slika 4.5 prikazuje postupak propagacije unatrag u MCTS algoritmu. Zelenom bojom prikazana je propagacija unatrag od čvora koji je proširen u koraku ekspanzije do korijena stabla. Nakon završetka ovog koraka, jedna iteracija MCTS algoritma je završena.



Slika 4.5: Postupak propagacije unatrag u MCTS algoritmu

4.2. Uvjet zaustavljanja i odabir najboljeg poteza

Algoritam se nastavlja izvršavati sve dok nije ispunjen uvjet zaustavljanja, npr. istek vremena ili dostignut broj iteracija. Nakon završetka željenog broja iteracija ili isteka vremena, odnosno zadanog uvjeta zaustavljanja, algoritam vraća najbolji potez koji je pronašao tijekom izvođenja. Najbolji potez obično je onaj koji je posjećen najviše puta tijekom izvođenja algoritma, a koji je pritom i najbolje ocijenjen. Može se dogoditi da najposjećeniji potez nije najbolje ocijenjen. U većini slučajeva najposjećeniji potez je i najbolje ocijenjen. Najbolje ocijenjeni potez je potez koji ima najveću prosječnu nagradu, odnosno najveći omjer između nagrade i broja posjeta čvoru, što odgovara komponenti prosječne vrijednosti čvora u UCB1 4.1 formuli.

5. Pregled zahtjeva

U ovom poglavlju definirani su funkcionalni i nefunkcionalni zahtjevi aplikacije. Prije same implementacije potrebno je definirati zahtjeve koje će aplikacija morati ispuniti. Precizno definirani zahtjevi ključni su za uspješnu implementaciju aplikacije. Iz precizno definiranih zahtjeva stvara se jasna slika o tome što se očekuje od aplikacije te se na temelju toga može pristupiti implementaciji. Precizno definirani zahtjevi olakšavaju podjelu zadatka na manje dijelove i omogućuju lakše praćenje napretka implementacije. Zahtjevi se mogu podijeliti na funkcionalne i nefunkcionalne. Funkcionalni zahtjevi definiraju konkretne funkcionalnosti koje aplikacija mora imati, dok nefunkcionalni definiraju kako aplikacija mora raditi.

5.1. Funkcionalni zahtjevi

Funkcionalni zahtjevi su glavna vodilja implementacije aplikacije, a to su:

- aplikacija mora biti implementirana kao mobilna aplikacija za operacijski sustav Android;
- aplikacija mora biti napisana po Model-View-ViewModel arhitekturi;
- aplikacija mora omogućiti igraču igranje igre protiv računala, tj. računalni igrač mora donositi smislene poteze;
- aplikacija mora omogućiti igraču igranje igre protiv drugog igrača na istom uređaju;
- računalni igrač mora koristiti MCTS algoritam za donošenje kvalitetnih poteza;
- aplikacija mora nuditi igraču preporuke za poteze koje bi mogao odigrati u trenutnom stanju igre;
- preporuke poteza moraju biti popraćene objašnjenjem;
- aplikacija mora omogućiti igraču odabir težine računalnog igrača;
- moraju postojati tri razine težine računalnog igrača: laka, srednja i teška;

– aplikacija mora nuditi adaptivnu težinu računalnog igrača, tj. računalni igrač mora slijediti igračevu razinu igre;
– aplikacija mora nuditi naprednu prilagodbu težine računalnog igrača, tj. ručno postavljanje parametara algoritma;
– aplikacija mora spremati povijest svih odigranih igara uključujući sve poteze;
– aplikacija mora omogućiti igraču uplitanje u tijek igre koju je igrao u prošlosti;
– aplikacija mora omogućiti brisanje povijesti igara;
– aplikacija korisniku mora omogućiti gašenje zvučnih efekata.

Navedeni zahtjevi jasno definiraju funkcionalnosti koje aplikacija mora sadržavati.

5.2. Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi definiraju kako aplikacija mora raditi, a to su:

– aplikacija mora biti kompatibilna s različitim verzijama operacijskog sustava Android;
– aplikacija mora biti prilagodljiva različitim veličinama zaslona;
– aplikacija mora biti stabilna i ne smije se rušiti;
– korisničko sučelje mora biti intuitivno i jednostavno za korištenje;
– aplikacija mora biti skalabilna, odnosno omogućiti jednostavno dodavanje novih funkcionalnosti, npr. novog algoritma za donošenje poteza;
– aplikacija mora imati prihvatljiv dizajn.

6. Odabir platforme za implementaciju

U ovom poglavlju opisan je razlog zašto je ova aplikacija implementirana kao mobilna aplikacija te zašto je *Android* odabran kao operacijski sustav.

6.1. Zašto mobilna aplikacija?

Mobilni uređaji postali su neizostavni dio životnog stila većine ljudi. Razni izvori navode da je broj korisnika pametnih telefona u svijetu početkom 2023. godine premašio 6.5 milijardi, dok se za 2025. godinu očekuje da će taj broj dostići 7 milijardi [12]. S obzirom na pristupačnost pametnih telefona, ovi uređaji postali su glavni medij za pristup internetu, društvenim mrežama, e-pošti, igrama, obrazovnim i drugim sadržajima. Pametni telefoni dostupni su tijekom cijelog dana za korištenje i omogućavaju korisnicima ublažiti dosadu i povećati produktivnost. Upravo zbog tog razloga, obrazovna aplikacija za igranje društvene igre „4 u nizu" implementirana je kao mobilna aplikacija.

6.2. Android kao operacijski sustav

6.2.1. Zašto Android?

Trenutno postoje dva popularna operacijska sustava za mobilne uređaje: *Android* i *iOS*. Uređaje s operacijskim sustavom *iOS* proizvodi isključivo *Apple*, dok uređaje s operacijskim sustavom *Android* proizvodi većina proizvođača mobilnih uređaja. S obzirom na navedeno, Android uređaji pružaju veću raznolikost u raznim aspektima, kao što su cijena, veličina ekrana, performanse i slično.

Nadalje, *Android* uređaji omogućuju korisnicima veću slobodu u preuzimanju, odnosno instaliranju aplikacija koje nisu dostupne u službenoj trgovini aplikacija, za razliku od *iOS* uređaja. Ova sloboda omogućava lakšu distribuciju aplikacije i testiranje aplikacije od strane korisnika prije nego što aplikacija postane dostupna u službenoj

trgovini aplikacija. Problem s objavom aplikacije u službenoj trgovini na iOS uređajima je ta što je potrebno platiti godišnju naknadu za objavljivanje aplikacije na *App Storeu*, koja iznosi 99 američkih dolara, dok je za *Google Play Store* potrebno platiti samo jednokratnu naknadu od 25 dolara [15]. Naime, uz nemogućnost instalacije aplikacija izvan službene trgovine i uz obvezu plaćanja godišnje naknade za objavljivanje aplikacija na App Storeu, troškovi razvoja aplikacije za *iOS* su znatno veći u odnosu na troškove razvoja aplikacije za *Android*.

Android je ujedno globalno najpopularniji operacijski sustav za mobilne uređaje. Prema podacima iz travnja 2024. godine, *Android* je na globalnom tržištu imao udio od 71.31%, dok je *iOS* imao udio od 27.95% [16].

6.2.2. O operacijskom sustavu Android

Android je operacijski sustav primarno razvijen za mobilne uređaje i tablete koji je razvila tvrtka *Google*. S vremenom, *Android* je postao popularan i na drugim uređajima, kao što su pametni televizori, pametni satovi i automobili, čime je stvoren jedan od najvećih ekosustava na svijetu. Sustav je baziran na jezgri *Linuxa* (engl. *Linux kernel*) i otvorenog je koda što znači da je dostupan svima za besplatno korištenje, prilagodbu i distribuciju. *Android Inc.* osnovali su Andy Rubin, Rich Miner, Nick Sears i Chris White 2003. godine u Kaliforniji, dok ga je 2005. godine otkupila tvrtka *Google* za 50 milijuna dolara [17]. Ovu kupnju mnogi smatraju jednom od najboljih investicija u povijesti *Googlea*, što je u 2010. godini izjavio i tadašnji potpredsjednik *Googlea*, David Lawee. Prvi uređaj s operacijskim sustavom *Android* bio je *HTC Dream* koji je predstavljen 2008. godine. U razdoblju od 2008. pa do danas, 2024. godine, *Android* je prošao kroz mnoge verzije, a najnovija verzija operacijskog sustava je *Android 14*. Svaka verzija operacijskog sustava donosi nove značajke, poboljšanja performansi i sigurnosne zakrpe.

Službeni jezik za razvoj aplikacija za *Android* je *Java*, no 2017. godine *Google* je uveo i podršku za programski jezik *Kotlin*. Nadalje, u svibnju 2019. godine, *Google* je objavio da je *Kotlin* preporučeni jezik za razvoj aplikacija za *Android* [11]. *Kotlin* je moderni programski jezik koji se izvodi na *JVM* (engl. *Java Virtual Machine*) i koji je kompatibilan i interoperabilan s programskim jezikom *Java*. Interoperabilnost *Kotlina* s *Javom* znači da se mogu koristiti u istom projektu, što je korisno za programere koji žele postupno prelaziti s *Jave* na *Kotlin*, odnosno koristiti stare *Java* biblioteke u novim *Kotlin* projektima.

Najpopularniji operacijski sustav za mobilne uređaje, Android, omogućava i ins-

taliranje aplikacija izvan službene trgovine korištenjem *APK* (engl. *Android Package*) datoteka. *APK* je arhivska datoteka koja sadrži sve potrebne datoteke za instalaciju aplikacije na *Android* uređaj. Sigurnost sustava temelji se na više koncepata [14], kao što su *App sandbox* [13], *App signing* i *Authentication*. *App sandbox* je koncept koji ograničava pristup aplikacije resursima i podacima drugih aplikacija, izvođenjem aplikacije u zasebnom okruženju.

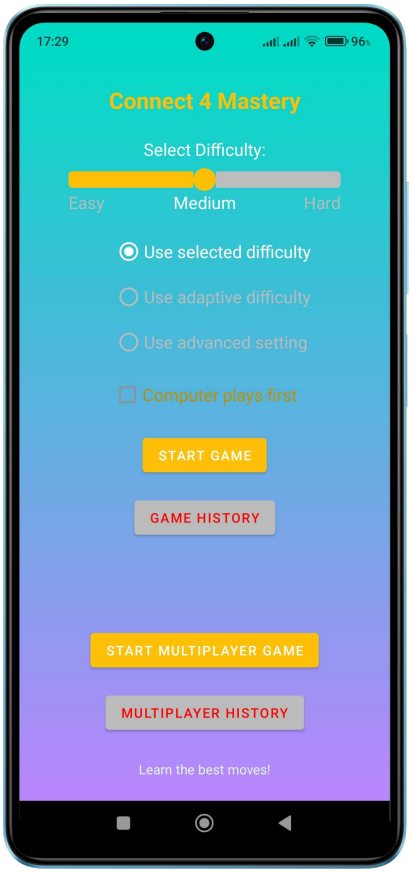
Ukratko, *Android* je globalno najpopularniji operacijski sustav za mobilne uređaje; omogućava veliku slobodu, dok pruža sigurnost i jeftine troškove razvoja aplikacija.

7. Pregled aplikacije

U sklopu ovog poglavlja opisan je rad aplikacije te najznačajnije funkcionalnosti koje ona pruža s uputama za korištenje. Ovo poglavlje je podijeljeno na tri dijela, a to su opis glavnog izbornika, povijest odigranih igara i zaslona za igru.

7.1. Opis glavnog izbornika

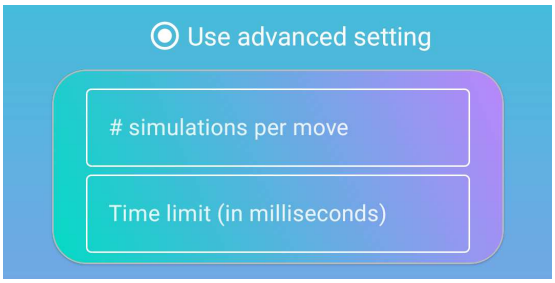
Pri pokretanju aplikacije korisniku je vidljiv tzv. *splash screen* koji prikazuje logotip aplikacije, a potom glavni izbornik aplikacije.



Slika 7.1: Glavni izbornik aplikacije

Na glavnom izborniku moguće je odabrati težinu računalnog igrača, postaviti koji igrač započinje igru te započeti igru (slika 7.1). Osim navedenog, moguće je i započeti igru protiv drugog igrača na istom uređaju. S ovog izbornika moguće je pristupiti povijesti odigranih igara, zasebno za *Singleplayer* i *Multiplayer* igre. Korisnik ima mogućnost birati između tri opcije predefiniranih težina računalnog igrača: laka, srednja i teška (engl. *easy, medium, hard*). Korisnik može odabrati jednu od ponuđenih opcija te započeti igru protiv računalnog igrača. U slučaju da korisnik ne želi igrati protiv računalnog igrača na predefiniranoj težini, može odabrati opciju *Use adaptive difficulty* koja omogućuje igranje protiv računalnog igrača koji prati korisnikovu razinu igre. Način zadavanja težine računalnog igrača pa tako i implementacija adaptivne težine opisani su u poglavlju 10. Korisnik može odabrati i opciju *Use advanced settings* koja omogućava prilagodbu parametara algoritma za donošenje poteza računalnog igrača.

Odabirom opcije *Use advanced settings* korisniku se prikazuju polja za unos parametara MCTS algoritma. Korisnik mora postaviti parametre *# of simulations* i *Time limit (ms)*, odnosno maksimalan broj simulacija koje će algoritam izvršiti za donošenje odluke te maksimalno vrijeme koje algoritam smije potrošiti za donošenje odluke izraženo u milisekundama. Postizanjem jednog od navedenih uvjeta, algoritam će donijeti odluku o potezu. Slika 7.2 prikazuje zaslon za postavljanje naprednih postavki.

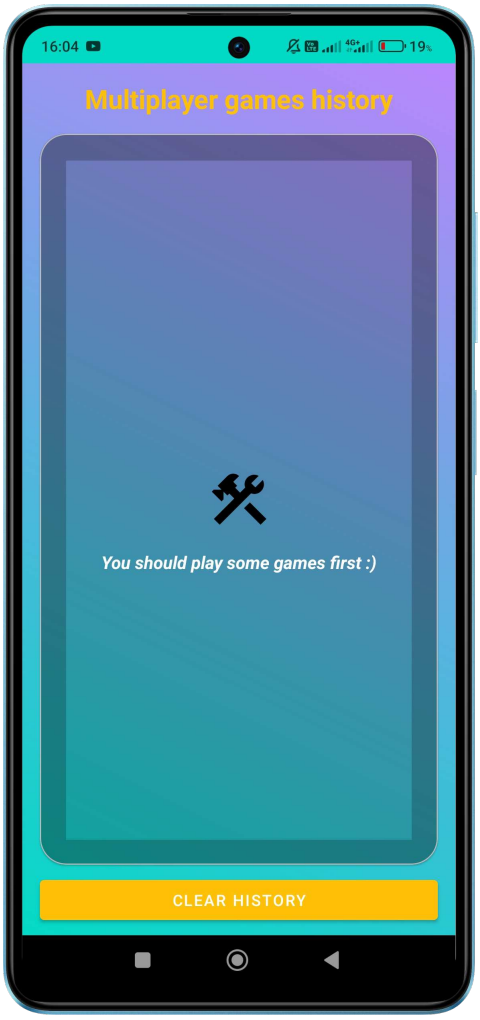


Slika 7.2: Zaslon za postavljanje naprednih postavki

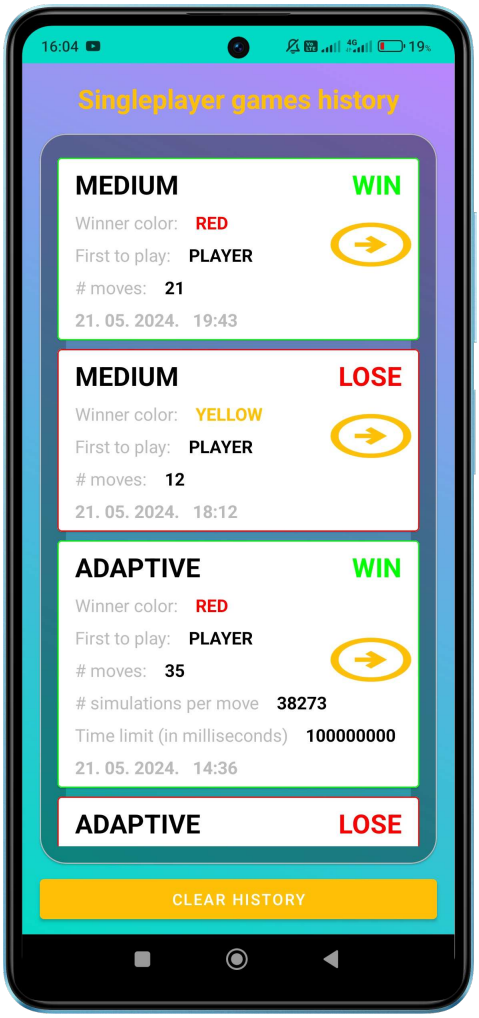
7.2. Povijest odigranih igara

Korisnik može pristupiti povijesti odigranih igara protiv računalnog igrača odabirom opcije *Game history* na glavnom izborniku, ali i povijesti odigranih igara protiv drugog igrača na istom uređaju odabirom opcije *Multiplayer history*. Povijest igara prikazuje se u obliku liste u kojoj je svaka stavka liste jedna odigrana igra. Slika 7.3 prikazuje zaslon povijesti igara kada nije odigrana niti jedna igra, dok slika 7.4 prikazuje isti zaslon kada je odigrana barem jedna igra.

Slika 7.4 prikazuje povijest igara odigranih protiv računalnog igrača. Za svaku



Slika 7.3: Prikaz povijesti igara kada nije odigrana niti jedna igra



Slika 7.4: Prikaz povijesti odigranih igara

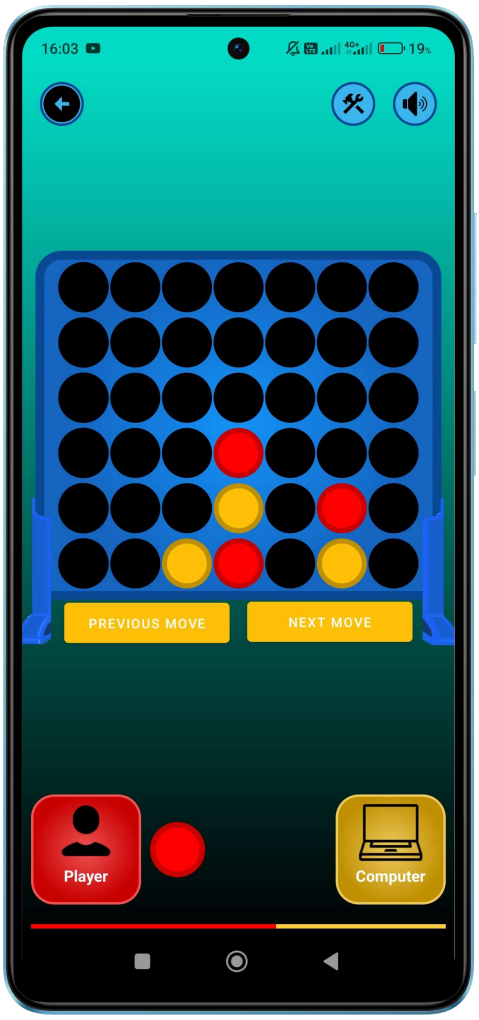
odigranu igru prikazuje se rezultat igre, težina računalnog igrača, tko je započeo igru, broj poteza i slično. U slučaju da korisnik nije igrao protiv računalnog igrača s već predefiniranom težinom, već protiv računalnog igrača s adaptivnom težinom, prikazuju se i parametri MCTS algoritma korišteni u toj igri. Najvažnija značajka povijesti igara je mogućnost pregleda svake odigrane igre. Klikom na jednu od odigranih igara prikazuje se zaslon s detaljima te igre i mogućnošću uplitanja u tijek igre. Detaljnije o ovoj mogućnosti bit će opisano u poglavlju 7.3.

Osim pregleda povijesti igara, korisnik može obrisati i cijelu povijest pritiskom na gumb *Clear history* te dodatnom potvrdom unutar dijaloga koji se prikazuje nakon pritiska na gumb.

7.3. Zaslon za igru

Dolazak na zaslon igre moguć je na dva načina, kroz glavni izbornik opisan u potpoglavlju 7.1 ili kroz zaslon povijesti odigranih igara opisan u potpoglavlju 7.2. Ovisno iz kojeg zaslona korisnik započinje igru, zaslon igre će izgledati različito. Slika 7.5 prikazuje zaslon igre kada korisnik dolazi iz glavnog izbornika, dok slika 7.6 prikazuje isti zaslon kada korisnik dolazi iz povijesti igara.

Slika 7.5: Zaslon za igru



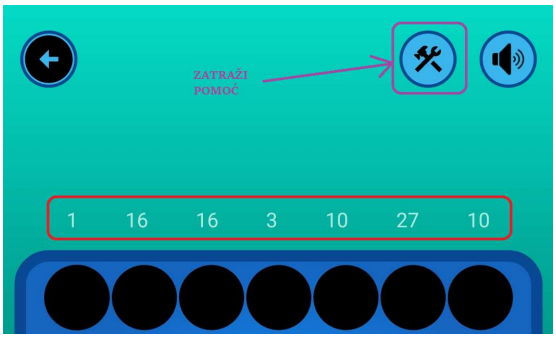
Slika 7.6: Zaslon za igru kada mu se pristupa iz povijesti igara

U slučaju da je korisnik pristupio igri kroz povijest nudi mu se mogućnost pregleda svakog poteza odigrane igre. Pregled poteza omogućava korisniku simuliaciju odigranih poteza pritiskom gumba *Next move* ili vraćanje na prethodni potez pritiskom gumba *Previous move*. Prilikom pregleda povijesti poteza, korisniku se daje mogućnost uplitanja u tijek igre tako što će umjesto pritiska na gumb *Next move* jednostavno

odabrati novi potez koji želi odigrati. Uplitanje u tijek igre omogućava korisniku promjenu poteza koji je odigrao i tako promijeni ishod igre. Ova značajka korisniku pruža mogućnost učenja iz vlastitih grešaka i poboljšavanje u igri.

Druga bitna značajka koja pomaže korisniku u učenju igre je mogućnost prikaza preporuka za poteze. Preporuke za poteze korisnik može zatražiti pritiskom gumba u gornjem desnom dijelu zaslona, odmah pokraj gumba koji dozvoljava uključivanje i isključivanje zvučnih efekata. Zvučni efekti odnose se na zvuk koji proizvode padajući tokeni prilikom dodira s pločom ili s drugim tokenima.

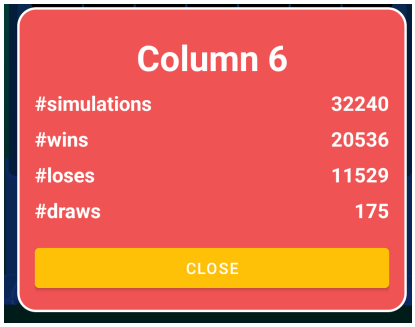
Preporuke se korisniku prikazuju u obliku brojčane vrijednosti (između -100 i 100) koja predstavlja koliko je određeni potez dobar, a nalaze se iznad svakog stupca ploče nakon što korisnik zatraži pomoć. Slika 7.7 prikazuje isječak zaslona igre s prikazanim preporukama za poteze.



Slika 7.7: Prikaz preporuka za poteze

Dodatno objašnjenje za iskazane brojčane vrijednosti može se dobiti pritiskom na tu vrijednost. Pritiskom na vrijednost otvara se dijalog koji kao objašnjenje daje statistiku simulacija koje su provedene za taj potez. Osim značaja preporuka za poteze, ova statistika pruža korisniku uvid u to kako algoritam donosi odluke te kako se ponaša u različitim situacijama. Zanimljivo je vidjeti razliku broja simulacija koje su provedene za različite poteze ovisno o njihovoj kvaliteti. Slika 7.8 prikazuje dijalog s objašnjenjem preporuke za potez.

S obzirom na to da proces donošenja odluka računalnog igrača i proces dobivanja kvalitete poteza može potrajati, u tijeku tog procesa korisniku je vidljiv indikator koji prikazuje napredak. Indikator je traka napretka (engl. *progress bar*) koja se prikazuje ispod igrača ako je zatražio pomoć ili ispod računalnog igrača ako je na potezu. Ova traka istovremeno ovisi o maksimalnom broju simulacija te o vremenskom ograničenju, ovisno o tome koji uvjet se prvi ispuni. Slika 7.9 prikazuje isječak zaslona igre s prikazanim indikatorom napretka.



Slika 7.8: Prikaz dijaloga s objašnjenjem preporuke za potez

Slika 7.9: Prikaz indikatora napretka

Na dnu zaslona vidljiva je boja pojedinog igrača i igrač koji je trenutno na potezu. Pozadina igrača odgovara boji tokena koju igrač koristi. Slika 7.5 prikazuje zaslon igre kada je na potezu igrač sa žutim tokenima, dok slika 7.6 prikazuje zaslon igre kada je na potezu igrač s crvenim tokenima. Kako se mijenja igrač na potezu, tako se uz animaciju mijenja pozicija i boja tokena koji se nalazi uz igrača. Ova animacija omogućuje korisniku da prati tko je na potezu te da se lakše orijentira u igri.

8. Arhitektura aplikacije

U ovom poglavlju opisana je struktura *Android* mobilne aplikacije za igranje društvene igre „4 u nizu”.

Aplikacija koristi strukturu sa samo jednim *activityjem* koji služi kao kontejner za različite *Fragmente*, čime se postiže lakša implementacija navigacije i bolja organizacija koda. Projekt je strukturiran po *Model-View-ViewModel* arhitekturi što omogućuje jasnu podjelu odgovornosti između različitih dijelova aplikacije, npr. logike igre, korisničkog sučelja i dohvaćanja podataka. Korisničko sučelje definirano je kroz XML (*Extensible Markup Language*), dok je programski jezik *Kotlin* korišten za implementaciju aplikacije.

8.1. MVVM arhitektura

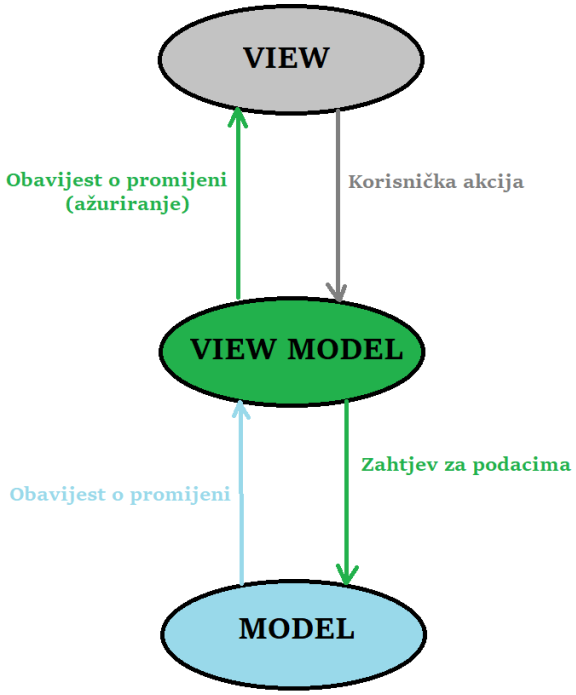
MVVM (*Model-View-ViewModel*) je najčešće korištena arhitektura, odnosno način organizacije koda u Android aplikacijama. Ova arhitektura omogućuje jasnu podjelu odgovornosti između različitih dijelova aplikacije. *Model* predstavlja podatke i poslovnu logiku aplikacije, odnosno pristupa podacima i obavlja poslovne operacije. *View* je taj koji prikazuje podatke korisniku, najčešće kroz XML datoteke. Iako korisnici započinju interakciju s aplikacijom preko *Viewa* koji šalje zahtjeve u *ViewModel*, *View* odgovor od *ViewModela* neće dobiti izravno.

ViewModel je posrednik između *Modela* i *Viewa*, odnosno sadrži logiku koja upravlja podacima i njihovom komunikacijom. Iako je *ViewModel* njihov posrednik, on nema referencu na *View* koji ga koristi. *ViewModel* je životno vezan uz *activity* ili *fragment* te se uništava s njima. Odvajanjem logike iz *Viewa* u *ViewModel* postiže se veća fleksibilnost te se olakšava testiranje koda. Također, prednost MVVM arhitekture je i to što se povećava mogućnost ponovne upotrebe koda i omogućava istovremeno programiranje više programera na istom projektu.

Jedna od popularnih arhitektura je i MVC (*Model-View-Controller*). Glavna razlika između MVC i MVVM arhitektura je to što MVC izravno povezuje *Model* i *View*

preko *Controllera*, dok MVVM arhitektura koristi *ViewModel* kao poveznicu između *Modela* i *Viewa*. Odvajanje poslovne logike od korisničkog sučelja značajno smanjuje kompleksnost koda i olakšava njegovo održavanje, što uvelike daje prednost MVVM nad MVC arhitekturom.

Slika 8.1 prikazuje MVVM arhitekturu.



Slika 8.1: MVVM arhitektura

- Poveznica između *View*a i *ViewModel*a prikazana je strelicom prema dolje i oznakom „Korisnička akcija". Korisnička akcija može biti bilo koja akcija koju korisnik izvrši na korisničkom sučelju, npr. pritisak na gumb, unos teksta u polje za unos i odabir stavke iz padajućeg izbornika.
- *ViewModel* obrađuje korisničku akciju i šalje odgovarajuće podatke *Modelu* ako za tim ima potrebe, što je na slici prikazano strelicom prema dolje i oznakom „Zahtjev za podacima".
- *Model* obrađuje zahtjev za podacima i šalje obavijest *ViewModelu* o promjeni podataka, što je na slici prikazano strelicom prema gore i oznakom „Obavijest o promjeni".

– *ViewModel* obrađuje nove podatke, provodi nad njima poslovnu logiku i obavještava *View* o promjeni podataka, što je na slici prikazano strelicom prema gore i oznakom „Obavijest o promjeni (ažuriranje)”. Ovaj mehanizam obavještavanja *Viewa* o promjeni podataka omogućuje *Viewu* ažuriranje i prikaz novih podataka korisniku, opisanih u potpoglavlju 9.1.

– *View* prikazuje nove podatke korisniku. Način na koji se *View* ažurira ovisi o korištenoj biblioteci. Dvije najčešće korištene biblioteke *Data Binding* i *View Binding* opisane su u potpoglavlju 9.2.

Primjer korisničke akcije prikazan je u isječku programskog koda 8.1. U ovom isječku, korisnik pritiskom na gumb *mbClearHistory* šalje zahtjev *ViewModelu* za brisanje povijesti igara i tu prestaje uloga *Viewa* u obradi tog zahtjeva. Za shvaćanje kako se zahtjev dalje obrađuje, potrebno je razumijeti kako *ViewModel* i *Model* komuniciraju, što je opisano u potpoglavlju 9.1.

```
1 mbClearHistory.setOnClickListener {
2     viewModel.clearHistory(args.multiplayerGames)
3 }
```

Kôd 8.1: Primjer korisničke akcije

8.2. Single-Activity arhitektura

U početku razvoja Android aplikacija, svaki zaslon u aplikaciji bio je predstavljen zasebnim *activityjem*. Ovaj pristup nije bio zadovoljavajuć zbog više razloga, među kojima su najvažniji:

– loše performanse - svako stvaranje novog *activityja* je skupo u smislu resursa, a i u smislu performansi;

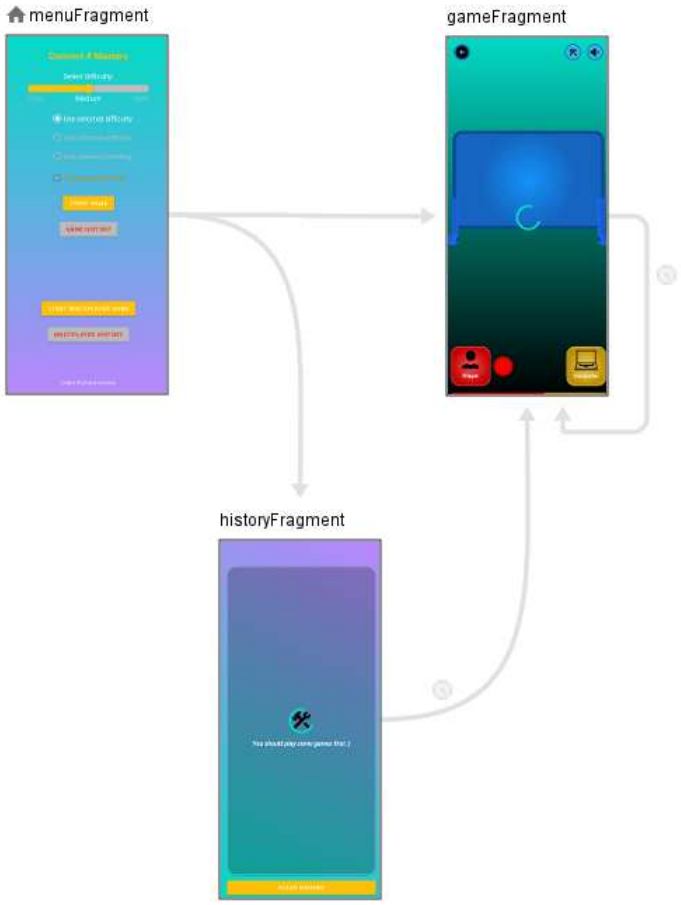
– složena navigacija koja dovodi do loše organizacije koda, posljedično i do težeg održavanja;

– složena komunikacija između *activityja*;

– kompleksnost upravljanja životnim ciklusom *activityja*.

Kako bi se riješili navedeni problemi, Google je predstavio *single activity* arhitekturu. Ova arhitektura znači da se cijela aplikacija sastoji od samo jednog *activityja* koji služi kao kontejner za različite *fragmente*. Navigacija između *fragmenata* odvija se uz pomoć *navigation* komponente [7]. *Navigation* komponenta omogućava jednostavnu navigaciju, slanje podataka kao i definiranje animacija prilikom prijelaza između

fragmenata. Navigacija se stoga svodi na zamjenu jednog *fragmenta* drugim, a ne na stvaranje novog *activityja*. Definiranje navigacije između *fragmenata* obavlja se u XML datoteci, no taj postupak vrlo je jednostavan s obzirom na to da razvojno okruženje *Android Studio* nudi vizualni alat za njenu izradu. Slika 8.2 prikazuje primjer navigacijskog grafa izrađenog kroz vizualni alat u *Android Studiu*.



Slika 8.2: Navigacijski graf

Korištenje jednog *activityja* umjesto više njih rješava ranije navedene probleme, a uz to pruža i bolju organizaciju koda i lakše održavanje aplikacije. Nadalje, *fragmenti* su modularni dijelovi aplikacije koji se mogu koristiti na više mjesta u aplikaciji, što dodatno povećava mogućnost ponovne uporabe koda.

Osim navigacije unaprijed, *navigation* komponenta omogućava i jednostavno definiranje navigacije unatrag, odnosno povratka na jedan od prethodnih *fragmenata*. Ovaj povratak moguće je ostvariti zbog postojanja *BackStacka* koji pamti prethodne *fragmente*.

9. Korištene biblioteke

U ovom poglavlju opisane su najznačajnije biblioteke korištene u razvoju aplikacije. Za kvalitetnu povezanost između korisničkog sučelja najznačajnije biblioteke su *Data Binding* i *View Binding*, dok se za upravljanje i obradu podataka koriste *ViewModel* i *LiveData*. Za pohranu podataka primjenjuje se *Room* biblioteka. Značajnu ulogu za efikasno izvođenje aplikacije ima i *Coroutines* biblioteka koja nudi mogućnost asinkronog izvođenja koda. Za ostvarenje Single-Activity arhitekture, opisane u potpoglavlju 8.2, najznačajnija je *Navigation* biblioteka.

9.1. ViewModel i LiveData

9.1.1. ViewModel

ViewModel je dio arhitekture koji sadrži poslovnu logiku i upravlja podacima, a životno je vezan uz *activity* ili *fragment* te se uništava s njima. *ViewModel* se ne instancira izravno, već se koristi *ViewModelProvider* klasa koja je zadužena za stvaranje, pohranu i povezivanje *ViewModela* s *activityjem* ili *fragmentom*. Pohrana *ViewModela* vrši se u *ViewModelStore* objektu koji je vezan uz *activity* ili *fragment*, odnosno svaki od njih ima svoj *ViewModelStore* objekt. Pri traženju *ViewModela*, *ViewModelProvider* prvo provjerava postoji li *ViewModel* za traženi *activity* ili *fragment* u *ViewModelStore* objektu, a ako ne postoji, stvara novi *ViewModel* i pohranjuje ga u *ViewModelStore* objekt. Tako je *ViewModel* vezan uz *activity* ili *fragment* te se uništava s njima.

ViewModel ne sadrži referencu na *View* koji ga koristi, već koristi *LiveData* objekte za obavještavanje *View*a o promjeni podataka. Primjer definiranja *LiveData* objekta u *ViewModelu* prikazan je u isječku programskog koda 9.1. Prvo je definiran *LiveData* objekt *_gamesLiveData* koji sadrži listu objekata tipa *Game*. Ovaj *LiveData* je privatni objekt, a pristup njemu ostvaren je preko javnog *gamesLiveData* objekta.

```
1 class HistoryViewModel(database: Connect4Database) : ViewModel() {
2
3     private val _gamesLiveData = MutableLiveData<List<Game>>()
```

```
    val gamesLiveData: LiveData<List<Game>> get() = _gamesLiveData
    ...
}
```

Kôd 9.1: Definiranje LiveData objekta u ViewModelu

Uz definiciju *LiveData* objekta, u *ViewModelu* definiraju se i metode koje obavljaju poslovnu logiku, npr. metoda *clearHistory()* koja briše povijest igara. Ova metoda poziva se iz *Viewa* (prikazano na primjeru u programskom kodu 8.1) nakon što korisnik pritisne gumb za brisanje povijesti igara. Ovisno o tome pregledava li korisnik povijest igara za jednog ili više igrača, metoda *clearHistory()* prima odgovarajući argument. Primjer implementacije metode *clearHistory()* prikazan je u isječku programskog koda 9.2.

```
fun clearHistory(multiplayerGames: Boolean) {
    viewModelScope.launch(Dispatchers.IO) {
        if(multiplayerGames) {
            gameDao.deleteMultiplayer()
        } else {
            gameDao.deleteSingleplayer()
        }
        _gamesLiveData.postValue(emptyList())
    }
}
```

Kôd 9.2: Implementacija metode clearHistory()

U ovom isječku, metoda *clearHistory()* koristi *viewModelScope* objekt za pokretanje nove *coroutine*. *Coroutine* su opisane u potpoglavlju 9.4, a ukratko rečeno, *coroutine* su laganija verzija dretvi (engl. *threads*) koje omogućuju asinkrono izvođenje koda. Asinkrono izvođenje koda omogućuje izvođenje dugotrajnih operacija, kao što je brisanje podataka iz lokalne *Room* baze podataka, bez blokiranja glavne niti. U ovom primjeru, metoda *clearHistory()* briše povijest igara iz lokalne baze podataka, a zatim postavlja vrijednost *_gamesLiveData* objekta na praznu listu. Postavljanje vrijednosti *_gamesLiveData* objekta na praznu listu obavještava pretplatnike *LiveData* objekta o promjeni podataka, što će rezultirati ažuriranjem *Viewa*. Način na koji se *View* pretplaćuje i ažurira ovisno o promjeni podataka opisan je u potpoglavlju 9.1.2.

9.1.2. LiveData

LiveData [6] objekti su objekti koji slijede oblikovni obrazac *Observer* i omogućuju *ViewModelu* obavještenje *Viewa* o promjeni podataka. Kako bi se *View* ažurirao ovisno o promjeni podataka, potrebna je pretplata *Viewa* na *LiveData* objekt. Pri pretplati se definira i životni ciklus pretplate, odnosno *View* je pretplaćen na *LiveData* objekt samo

dok je *activity* ili *fragment* u aktivnom stanju. Aktivno stanje predstavlja stanje u kojem je životni ciklus [5] *activityja* ili *fragmenta* u stanju *STARTED* ili *RESUMED*. Tako će se *View* ažurirati samo ako je u aktivnom stanju, što je važno za učinkovito upravljanje resursima.

U programskom kodu 8.1 prikazan je primjer korisničke akcije u kojem korisnik pritiskom na gumb šalje zahtjev *ViewModelu* za brisanje povijesti igara. Kako bi se *View* ažurirao o promjeni podataka, potrebno ga je pretplatiti na *LiveData* objekt koji sadrži relevantne podatke. Definicija *LiveData* objekta u *ViewModelu* izložena je u isječku programskog koda 9.1. Primjer pretplate na *LiveData* objekt kroz *View* prikazan je u isječku programskog koda 9.3.

```
viewModel.gamesLiveData.observe(viewLifecycleOwner) { games ->
    games?.let {
        historyAdapter.submitList(it)

        //set visibility for tvEmptyHistory and ivEmptyHistory
        if (it.isEmpty()) {
            binding.tvEmptyHistory.visibility = View.VISIBLE
            binding.ivEmptyHistory.visibility = View.VISIBLE
        } else {
            binding.tvEmptyHistory.visibility = View.GONE
            binding.ivEmptyHistory.visibility = View.GONE
        }
    }
}
```

Kôd 9.3: Pretplata na LiveData objekt

U ovom isječku programskog koda, *View* se pretplaćuje na *LiveData* objekt *gamesLiveData* koji sadrži listu igara. Pretplata se vrši pozivom metode *observe()* nad *LiveData* objektom, a kao argumenti metode *observe()* predaju se *LifecycleOwner* i akcija koja se izvršava kada dođe do promjene podataka. U slučaju ovog primjera, akcija koja se izvršava kada dođe do promjene podataka je prikazivanje povijesti igara u *RecyclerViewu* i postavljanje vidljivosti *TextViewa* i *ImageViewa* ovisno o tome je li povijest igara prazna ili ne.

Također, način pristupa elementima korisničkog sučelja u *Viewu* prikazan je kroz *binding* objekt, korištenjem biblioteke *View Binding* opisane u potpoglavlju 9.2.

9.2. View Binding i Data Binding

U počecima razvoja Android aplikacija, pristup elementima korisničkog sučelja odvijao se kroz metodu *findViewById()*. Ovakav pristup je bio prilično neefikasan jer je za

svaki element korisničkog sučelja potrebno pozvati metodu *findViewById()* kako bi se dohvatila referenca na taj element. Ovakav pristup ne samo da je neefikasan i loše čitljiv, već je i sklon greškama, stoga je u sklopu Android Jetpack biblioteke predstavljena *View Binding* biblioteka [9]. Osim *View Bindinga*, postoji i *Data Binding* biblioteka [4] koja omogućava povezivanje podataka s korisničkim sučeljem. U nastavku su opisane obje biblioteke i način na koji se koriste u Android aplikacijama.

Primjer pristupa *TextView* elementu korisničkog sučelja i postavljanju njegovog teksta kroz *findViewById()* metodu uočljiv je u isječku programskog koda 9.4.

```
1 findViewById<TextView>(R.id.tvTitle).apply {
2     text = viewModel.title
3 }
```

Kôd 9.4: Pristup elementu korisničkog sučelja kroz findViewById()

9.2.1. View Binding

View Binding biblioteka pruža pristup elementima korisničkog sučelja bez potrebe za pozivanjem metode *findViewById()*. Postavljanje ove biblioteke vrlo je jednostavno i sastoji se od dodavanja linije koda u *build.gradle* datoteku. Detaljnije upute za postavljanje i korištenje *View Bindinga* mogu se pronaći na službenoj stranici biblioteke [9].

Primjer pristupa *TextView* elementu korisničkog sučelja bez korištenja *View Binding* biblioteke izložen je u isječku programskog koda 9.4. Isti taj programski odsječak, ali s korištenjem *View Binding* biblioteke, prikazan je u isječku programskog koda 9.5. Kao što je vidljivo iz tog isječka, pristup elementima korisničkog sučelja je jednostavniji i čitljiviji te manje sklon greškama.

```
1 binding.tvTitle.text = viewModel.title
```

Kôd 9.5: Pristup elementu korisničkog sučelja kroz View Binding

9.2.2. Data Binding

Data Binding biblioteka omogućuje povezivanje podataka s korisničkim sučeljem izravno u XML datoteci. Postavljanje *Data Bindinga* u projekt je jednostavno, a detaljno je objašnjeno na službenoj stranici ove biblioteke [4].

Primjer pristupa *TextView* elementu korisničkog sučelja bez korištenja *View Binding* biblioteke prikazan je u isječku programskog koda 9.4. Isti odsječak, ali korištenjem *Data Binding* biblioteke, prikazan je u isječku programskog koda 9.6. U ovom

odsječku, vidljivo je kako se *TextView* elementu *tvTitle* postavlja tekst kroz *Data Binding* biblioteku.

```
<TextView
    android:id="@+id/tvTitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{viewModel.title}"  <--------------------
/>
```

Kôd 9.6: Postavljanje teksta u TextView element korisničkog sučelja kroz Data Binding

Na ovaj način povećava se čitljivost koda za neki *fragment* ili *activity* jer se logika za postavljanje podataka u korisničko sučelje odvaja od ostale logike. U slučaju složenijeg programskog koda koji bi se odnosio na konfiguraciju korisničkog sučelja, taj kôd bi se teško ili nikako mogao izdvojiti izravno u XML datoteku. Takav kôd se može izdvojiti iz *activityja* ili *fragmenta* uz pomoć *Binding Adaptera* i time dodatno povećati čitljivost koda. *Binding Adapter* može se opisati kao statička metoda koja služi za prilagodbu podacima za prikaz u korisničkom sučelju, a označava se anotacijom *@BindingAdapter*. Primjer *Binding Adaptera* korištenog za definiranje boje teksta za argument koji predstavlja pobjednika igre prikazan je u isječku programskog koda 9.7. Ovaj *Binding Adapter* koristi se u XML datoteci za postavljanje boje teksta ovisno o tome je li igra završila neriješeno, pobjedom igrača 1 ili pobjedom igrača 2.

```
// u zasebnoj datoteci
@JvmStatic
@BindingAdapter("android:setWinnerColor")
fun TextView.setWinnerColor(winnerColor: String) {
    val color = when (winnerColor) {
        "1" -> ContextCompat.getColor(context, R.color.red)
        "2" -> ContextCompat.getColor(context, R.color.yellow)
        else -> ContextCompat.getColor(context, R.color.white)
    }
    this.setTextColor(color)
}

// u XML datoteci
<TextView
    android:id="@+id/tvWinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:setWinnerColor="@{String.valueOf(viewModel.winner)}" <--------------------
    ...
/>
```

Kôd 9.7: Definiranje i korištenje Binding Adaptera

Prednost korištenja *Data Binding* biblioteke u odnosu na *View Binding* je ta što *View* može izravno pristupiti *LiveData* objektima iz *ViewModela* i automatski ažurirati

korisničko sučelje nakon promjene podataka. Nije potrebno ručno pretplaćivanje na *LiveData* objekte, već se automatski obavlja kroz *Data Binding* biblioteku kada se koristi *LiveData* objekt u XML datoteci. No, prednost *View Bindinga* u odnosu na *Data Binding* svakako je jednostavnost korištenja i brzina kompajliranja aplikacije.

9.3. Room

Room [8] je biblioteka koja pruža jednostavnu izradu i upravljanje lokalnom bazom podataka, a temelji se na SQLite bazi podataka. *Room* biblioteka sastoji se od tri glavna dijela:

- *Database* - predstavlja bazu podataka i sadrži metode za pristup *DAO* objektima.
- *Entity* - predstavlja tablicu u bazi podataka, a svaki objekt klase *Entity* predstavlja jedan redak u tablici.
- *DAO* (*Data Access Object*) - sadrži metode za upravljanje podacima u bazi podataka.

Primjer definicije *Entity* klase prikazan je u isječku programskog koda 9.8. U navedenom je isječku definirana klasa *DbGame* koja predstavlja tablicu *games* u bazi podataka. Svaki objekt klase *DbGame* predstavlja jedan redak u tablici *games* i sadrži atribute kao što su *idGame*, *moves* i *winner*. Primarni ključ tablice *games* označen je anotacijom *@PrimaryKey*, dok je automatsko generiranje vrijednosti za taj atribut označeno anotacijom *@PrimaryKey(autoGenerate = true)*. Anotacija *@ColumnInfo* se koristi za definiranje imena stupca u tablici, dok se anotacija *@SerialName* upotrebljava za definiranje imena atributa u JSON formatu koje se koristi za serijalizaciju i deserijalizaciju objekta.

```
1
2  @Entity(tableName = "games")
3  @Serializable
4  data class DbGame(
5      @PrimaryKey(autoGenerate = true)
6      @SerialName("idGame")
7      @ColumnInfo(name = "idGame")
8      val idGame: Int,
9
10     @SerialName("moves")
11     @ColumnInfo(name = "moves")
12     val moves: List<Move>,
13
14     @SerialName("winner")
15     @ColumnInfo(name = "winner")
```

```
    val winner: Int,

    //... ostali atributi
)
```

Kôd 9.8: Definiranje Entity klase

Primjer definicije sučelja *DAO* prikazan je u isječku programskog koda 9.9 koji definira sučelje *GameDao*. Navedeno sučelje sadrži metode za upravljanje podacima u tablici *games*. Metoda *insert()* koristi se za dodavanje novog retka u tablicu *games*, a metoda *deleteSingleplayer()* za brisanje svih redaka u tablici *games* u kojima je igrač igrao protiv računala.

```
@Dao
interface GameDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(ride: DbGame): Long

    @Query("DELETE FROM games WHERE humanPlayer != 0")
    fun deleteSingleplayer()

    //... ostale metode
}
```

Kôd 9.9: Definiranje DAO sučelja

Primjer korištenja *DAO* sučelja prikazan je u isječku programskog koda 9.2. Važno je naglasiti da se pristup *DAO* objektima vrši isključivo asinkrono, a to se postiže korištenjem *coroutina*. Pokušaj pristupa *DAO* objektima iz glavne niti nije dozvoljen jer bi blokirao glavnu nit i tako zamrznuo korisničko sučelje. Stoga, prema konfiguraciji, *Room* će baciti iznimku pri pokušaju pristupanja *DAO* objektima iz glavne niti.

Od ostalih značajki *Room* biblioteke, važno je spomenuti i mogućnost definiranja relacija između tablica, mogućnost definiranja indeksa za brži pristup podacima, kao i mogućnost definiranja migracija baze podataka. Provjera ispravnosti upita u *Room* biblioteci vrši se već pri kompajliranju aplikacije, što znači da se greške otkrivaju već u ranoj fazi, a ne tek pri izvođenju upita.

U definiranom entitetu *DbGame*, u isječku programskog koda 9.8, nalazi se atribut *moves* koji predstavlja listu poteza u igri. Budući da *List<Move>* nije jedan od osnovnih tipova podataka, potrebno je definirati konverter koji će omogućiti pohranu liste objekata tipa *Move* u bazu podataka. Primjer definicije konvertera prikazan je u isječku programskog koda 9.10. U ovom isječku programskog koda, definirana klasa *RoomConverters* sadrži metode koje omogućuju serijalizaciju i deserijalizaciju liste objekata tipa *Move*.

```
class RoomConverters {
    private val json = Json { ignoreUnknownKeys = true }

    @TypeConverter
    fun movesToJson(moves: List<Move>?): String? {
        return moves?.let { json.encodeToString(it) }
    }

    @TypeConverter
    fun jsonToMoves(jsonString: String?): List<Move>? {
        return jsonString?.let { json.decodeFromString(it) }
    }
}
```

Kôd 9.10: Definiranje Type Convertera

Osim definiranja konvertera, potrebno je i registrirati konverter u bazi podataka. Registracija konvertera u bazi podataka vrši se anotacijom *@TypeConverters*, a kao argument anotacije predaje se klasa koja sadrži konvertere. Definiranje Room baze podataka, pridruživanje entiteta i registracija konvertera prikazana je u isječku programskog koda 9.11. Također, vidljivo je kako se pristup *DAO* objektu vrši kroz apstraktnu metodu *gameDao()*.

```

@Database(entities = [DbGame::class], version = 1, exportSchema = false)
@TypeConverters(RoomConverters::class)
abstract class Connect4Database : RoomDatabase() {
    abstract fun gameDao(): GameDao

    companion object {
        @Volatile
        private var INSTANCE: Connect4Database? = null

        fun getDatabase(context: Context): Connect4Database {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context,
                    Connect4Database::class.java,
                    "connect4databasev0.0.4"
                ).addCallback(object : Callback() {
                }).build()

                INSTANCE = instance
                instance
            }
        }
    }
}
```

Kôd 9.11: Definiranje Room baze podataka

Jednom kada je *Room* baza podataka definirana, njeno korištenje vrlo je jednostavno. Ipak, postavljanje same baze podataka može biti zahtjevno. Problem pri postavljanju ove baze podataka u Android projekt može nastati već pri umetanju potrebnih biblioteka u *build.gradle* datoteku. Naime, *Room* biblioteka ovisi o drugim bibliotekama iz *Android Jetpack* skupa biblioteka, a verzije tih biblioteka moraju biti usklađene. Ako verzije nisu usklađene, može doći do grešaka pri kompajliranju, ali i pri izvođenju aplikacije. Ovisnost o drugim bibliotekama ima i svoje prednosti, kao što je dobra integracija primjerice s *LiveData* bibliotekom, što omogućuje praćenje promjena u bazi podataka i automatsko ažuriranje korisničkog sučelja.

9.4. Coroutines

Asinkrono izvođenje koda u Android aplikacijama može se postići korištenjem *coroutines* biblioteke [3]. *Coroutines* je jedna od standardnih biblioteka programskog jezika Kotlin i omogućuje asinkrono izvođenje koda bez potrebe za ručnim korištenjem dretvi. Mnogi jezici upotrebljavaju ključne riječi poput *async* i *await* za asinkrono izvođenje koda, što je često nepregledno i teško za održavanje. *Coroutines* omogućuju asinkrono izvođenje koda na jednostavan i čitljiv način, a pritom ne blokiraju glavnu nit. Blokiranje glavne niti aplikacije može, a učestalo i rezultira, zamrzavanjem korisničkog sučelja. Stoga su *coroutine* preporučeni mehanizam za asinkrono izvođenje koda u Android aplikacijama.

Primjer korištenja *coroutines* biblioteke već je prikazan u isječku programskog koda 9.2. U tom isječku programskog koda, metoda *clearHistory()* koristi *viewModelScope* objekt za pokretanje nove *coroutine* koji briše povijest igara iz lokalne baze podataka. Svaka *coroutina* mora biti pokrenuta unutar nekog *CoroutineScope* objekta, a *viewModelScope* objekt je jedan od predefiniranih *CoroutineScope* objekata koji je vezan uz *ViewModel*. Tako će se *coroutina* automatski zaustaviti s *ViewModelom*, što je ključno za efikasno upravljanje resursima. Osim *viewModelScope* objekta, postoji i *lifecycleScope* objekt koji je vezan uz životni ciklus *activityja* ili *fragmenta*. Koristi se za pokretanje *coroutina* koje se zaustavljaju s *activityjem* ili *fragmentom*. Prednost je i ugrađena podrška za prekidanje *coroutina*, što je korisno za prekidanje dugotrajnih operacija.

10. Implementacijski detalji

U ovom poglavlju opisani su detalji implementacije aplikacije kao što je implementacija ploče za igru, implementacija padajućeg tokena, implementacija težine računalnog igrača i slično.

10.1. Implementacija ploče za igru

Igrača ploča (slika 2.1) implementirana je kao *LinearLayout* definiran u *XML* datoteci. Dodavanje stupaca i redaka na ploču vrši se dinamički u *Kotlin* kodu. Svaki stupac je zapravo *LinearLayout* koji sadrži *ImageView* elemente kružnog oblika. Svaki *ImageView* element predstavlja jedan token ili prazno mjesto na ploči. Prazna mjesta na ploči označena su crnom bojom, a tokeni su crvene i žute boje. Pri dinamičkom dodavanju stupaca na ploču, svakom stupcu dodaje se *OnTouchListener* koji omogućava igranje potrebnih poteza. Kada igrač odabere i dodirne stupac u koji želi postaviti token, slobodna mjesta u tom stupcu će se prikazati svijetlo sivom bojom, a mjesto na koje će njegov token pasti će se obojiti bojom koja odgovara njegovoj boji tokena. Potez se ne izvršava sve dok igrač ne podigne prst s ekrana, čime se token postavlja na odabrano mjesto. Korisnik ima priliku odustati od poteza tako da podigne prst s ekrana na mjestu koje ne odgovara stupcu u kojem je odabrao igrati. Ova funkcionalnost korisniku omogućava lakšu vizualizaciju svog poteza i novog stanja ploče prije nego što se potez izvrši. Slika 10.1 prikazuje igraču ploču s navedenom funkcionalnošću.

Jednom kada korisnik odabere stupac u koji želi postaviti token, pritisne ga te podigne prst s ekrana nad tim istim stupcem, potez se izvršava tako da token pada na najniže slobodno mjesto u odabranom stupcu. Način implementacije padajućeg tokena opisan je u potpoglavlju 10.2.

Slika 10.1: Igrača ploča igre „4 u nizu” s odabranim potezom

10.2. Implementacija padajućeg tokena

Padajući token implementiran je kao *ImageView* element koji po obliku i veličini odgovara praznom mjestu na ploči, a po boji odgovara boji igračevog tokena. Za vrijeme između poteza, dok se korisnik ne odluči za potez, padajući token se ne prikazuje, već je nevidljiv. Kada korisnik odabere stupac u koji želi postaviti token, padajući token se prikaže na vrhu odabranog stupca i počne padati prema dnu stupca. No prije samog prikazivanja padajućeg tokena, potrebno je postaviti pravu boju tokena, postaviti ga na pravu poziciju te izračunati konačnu poziciju na kojoj će token završiti. Padanje tokena implementirano je kao animacija koja se izvršava unutar 750 milisekundi uz efekt odskoka na dnu stupca, a popraćena je i zvukom. Pri završetku animacije, polje na kojem je završio padajući token oboji se u boju tokena, a padajući token postane nevidljiv. Na ovaj način isti postupak se ponavlja za svaki potez korištenjem istog *ImageView* elementa.

10.3. Implementacija težine računalnog igrača

Pokretanje igre protiv računalnog igrača korisniku omogućuje odabir preddefinirane težine: *„lako”*, *„srednje”* i *„teško”*. Korisnik može odabrati računalnog igrača koji će se prilagoditi njegovoj razini igranja ili ručno postaviti parametre računalnog igrača. Detalji su opisani u poglavlju 7.

S obzirom na to da je implementacija računalnog igrača temeljena na MCTS algoritmu, težina računalnog igrača određena je brojem simulacija koje će se izvršiti prije svakog poteza. Za najlakšu preddefiniranu težinu računalnog igrača, broj simulacija postavljen je na 1000, za srednju težinu na 10000, a za najtežu težinu na 100000

simulacija.

Velika je razlika u kvaliteti poteza između svakog od navedenih težina računalnog igrača te upravo zbog toga korisnik može odabrati težinu računalnog igrača ovisno o svojoj razini igranja. Kao i kod preddefiniranih težina, razina kvalitete poteza određena je brojem simulacija koje će se izvršiti prije svakog poteza. Pri završetku igre, računa se novi broj simulacija koje će algoritam izvršiti prije svakog poteza, a taj broj simulacija postaje novi parametar za budućeg računalnog igrača na adaptivnoj težini. Model po kojem se određuje nova težina adaptivnog igrača vrlo je jednostavna te ovisi samo o trenutnom parametru težine (broju iteracija algoritma), ukupnom broju odigranih igara i ishodu igre. Ukupni broj odigranih igara koristi se za bržu prilagodbu računalnog igrača korisnikovoj razini igranja. Nakon više odigranih igara, težina računalnog igrača postaje stabilnija i ne mijenja se tako drastično kao kada igrač ima mali broj odigranih igara. Faktor s kojim će se težina računalnog igrača mijenjati ovisi o formuli 10.1.

$$\text{faktor} = 1.05 + e^{-0.05 \cdot \text{broj\_odigranih\_igara}} \tag{10.1}$$

Ovisno o ishodu igre, težina računalnog igrača može se povećati ili smanjiti. U slučaju korisnikove pobijede, trenutna težina (broj iteracija) računalnog igrača množi se s faktorom iz formule 10.1. U slučaju korisnikovog poraza, trenutna težina računalnog igrača dijeli se s tim faktorom. Na ovaj se način računalni igrač prilagođava korisnikovoj razini igranja, čime se postiže bolje iskustvo igranja i učenja kroz igru.

Za naprednije korisnike koji žele ručno postaviti parametre računalnog igrača, osim broja iteracija moguće je postaviti i maksimalno vrijeme izvršavanja algoritma. Tako se može postići različit stil igranja računalnog igrača, s obzirom na to da iteracije u kasnijem dijelu igre zahtijevaju znantno manje vremena za izvršavanje.

11. Usporedba s drugim varijantama algoritma

U sklopu ovog rada su implementirana i dva druga algoritma za donošenje preporuka za poteze u igri „4 u nizu". Osim MCTS-a implementiran je i temelj tog algoritma, slučajno donošenje poteza bez izgradnje stabla pretrage. Drugi implementirani algoritam je hibridni pristup koji kombinira MCTS i minimax u koraku simulacije MCTS algoritma.

11.1. Slučajno donošenje poteza

U svrhu usporedbe s MCTS-om, implementiran je algoritam koji donosi nasumične poteze bez izgradnje stabla pretrage. Veliki broj simulacija u kojima se donose nasumični potezi jedan je od temelja MCTS algoritma. Monte Carlo metoda povijesno je korištena za simulacije i donošenje odluka u raznim područjima, a napredak računalne tehnologije dodatno je skrenuo pozornost na ovu metodu.

Konceptualno, algoritam koji donosi nasumične poteze vrlo je jednostavan kao i implementacija. Algoritam se sastoji od glavne metode čiji je cilj donijeti odluku o potezu. U ovoj metodi izvodi se petlja koja se ponavlja sve dok se ne odigra određeni broj simulacija. Svaka iteracija petlje predstavlja jednu simulaciju igre, gdje jedna simulacija igre predstavlja odigravanje igre slučajnim potezima iz trenutnog stanja do kraja igre. Nakon što se odigra određeni broj simulacija, algoritam donosi odluku o potezu na temelju statistike koja je prikupljena tijekom simulacija. Odluka o potezu donosi se tako da se odabere stupac u kojem je najveći broj pobjeda, najmanji broj poraza ili neki drugi kriterij. U ovom radu kriterij za donošenje poteza je razlika između broja pobjeda i broja poraza za svaki stupac.

Ovaj algoritam nije se pokazao dobrim za igru „4 u nizu" zbog više razloga, no svakako je u prednosti nad potpuno nasumičnim odabirom poteza. U deset simuliranih igara protiv ovog algoritma, MCTS je pobijedio u svih deset bez obzira na broj simula-

cija koje su se izvršile (minimalno 100 simulacija). Porastom broj simulacija, razlika u kvaliteti poteza između ova dva algoritma postaje sve veća. Trajanje izvođenja ovog algoritma za isti broj simulacija je znatno manje od izvođenja MCTS-a, ali kvaliteta poteza je znatno lošija. U slučaju da je kriterij zaustavljanja vrijeme izvođenja, čak niti dvostruko više vremena u odnosu na MCTS algoritam nije dovoljno za postizanje kvalitete poteza koju MCTS postiže.

Značaj izgradnje stabla uistinu je velik. Postojanje stabla pomaže usmjeriti istraživanje prostora stanja u smjeru koji se čini najboljim i to na mnogobrojnim razinama stabla. Bez korištenja stabla, u slučaju ovog algoritma, kada su svi stupci i dalje otvoreni za potez, šest od sedam poteza otići će u stupac kojem ne pripada najbolji mogući potez. To proizlazi iz činjenice da je vjerojatnost nasumičnog odabira svakog od stupaca jednaka, a ne temeljena na kvaliteti poteza. Na konkretnom primjeru, MCTS algoritam s konstantom istraživanja $C = 1.4$ u 30000 iteracija prilikom odabira trećeg poteza igre, za istraživanje će odabrati stupac koji se čini najboljim u više od 50% svih iteracija.

11.2. Hibridni algoritam

Hibridni algoritam koji je opisan u ovom potpoglavlju kombinacija je MCTS-a i minimax algoritma. Uporaba minimax algoritma na velikom prostoru stanja pretežno nije najbolje rješenje zbog velikog broja mogućih poteza, što ujedno zahtijeva veliku količinu memorije i više vremena za izvođenje.

Osnovna verzija MCTS algoritma koristi nasumično donošenje poteza u fazi simulacije. Prednost nasumične simulacije je jednostavnost implementacije, brzina izvođenja, smanjena pristranost i sposobnost pretraživanja velikog prostora stanja s nepotpunom informacijom. Sve ovo omogućuje osnovnoj verziji MCTS algoritma korištenje u širokom spektru problema. No, nasumično donošenje poteza ima i svoje nedostatke kao što su sporo konvergiranje prema optimalnom rješenju, potreba za velikim brojem simulacija i potreba za velikom količinom memorije. Ishod simulacija može biti vrlo nepouzdan i često ne odražava stvarnu kvalitetu poteza. U poglavlju 7 opisan je postupak kojim korisnik može zatražiti prijedlog poteza. Prije donošenja odluke, korisnik prijedlog poteza može zatražiti više puta, što pruža mogućnost uočavanja visoke varijabilnosti kvalitete poteza.

Hibridnim pristupom kombiniraju se prednosti MCTS-a i minimaxa. Implementacija koristi MCTS za izgradnju stabla pretrage, dok se u fazi simulacije dodatno služi minimaxom s dubinom pretrage postavljenom na dva. Ovaj pristup omogućava

donošenje kvalitetnijih poteza u fazi simulacije, dok u teoriji pridonosi i bržoj konvergenciji prema optimalnom rješenju. U slučaju kada se unutar zadane dubine pretrage minimaxa pronađe pobjednički ili gubitnički potez, prestaje simulacija te slijedi propagacija rezultata prema korijenu stabla. Ako se unutar zadane dubine pretrage ne pronađe pobjednički ili gubitnički potez, kao i u klasičnom MCTS-u, donose se nasumični potezi do kraja igre.

Ovaj algoritam pokazuje bolje rezultate protiv klasičnog MCTS-a u jednakom broju simulacija. U deset simuliranih igara (koje nisu završile izjednačeno), hibridni algoritam pobijedio je u čak sedam igara protiv klasičnog MCTS-a. No, u slučaju kada se umjesto broja simulacija koristi vrijeme izvođenja kao kriterij zaustavljanja i ako se u obzir ne uzimaju igre koje su završile izjednačeno, klasični MCTS algoritam je pobijedio u osam od deset igara. Ova razlika proizlazi iz činjenice da je hibridni algoritam sporiji od klasičnog MCTS-a. Za donošenje istog poteza u istoj fazi igre, hibridnom pristupu treba gotovo dvostruko više vremena u osnosu na klasični MCTS algoritam. Na primjeru odabira četvrtog poteza u igri, klasični MCTS algoritam napravio je 63000 iteracija, dok se u isto vrijeme hibrnidnim pristupom napravilo 30000 iteracija.

Stoga, hibridni algoritam pokazuje bolje rezultate u jednakom broju simulacija, dok je klasični MCTS bolji u jednakom vremenu izvođenja. Uzimajući u obzir da je klasični MCTS jednostavniji za implementaciju, brži i kvalitetniji u istom vremenu izvođenja, hibridni pristup nije preporučljiv za korištenje u igri „4 u nizu”.

12. Moguće nadogradnje i poboljšanja

U ovom poglavlju opisane su nadogradnje i poboljšanja aplikacije koje bi se mogle implementirati u budućnosti. Navedene nadogradnje i poboljšanja ne odnose se samo na kvalitetu poteza računalnog igrača, već i na korisničko sučelje, iskustvo igranja i slično.

Postoje razni algoritmi koji se mogu koristiti za donošenje kvalitetnih poteza koje računalni igrač može izvršiti. Jedan takav algoritam opisan je u poglavlju 11, a to je modifikacija MCTS algoritma koja u koraku simulacije uključuje i *minimax* algoritam. Ovaj algoritam poboljšava kvalitetu poteza računalnog igrača, no zahtjeva i više resursa za izvršavanje. Kao poboljšanje aplikacije, moguće je ponuditi korisniku odabir ovog algoritma za igranje protiv računalnog igrača.

Trenutno, kao objašnjenje kvalitete poteza koje aplikacija predlaže korisniku, koristi se statistika kao što je posjećenost čvorova u stablu pretrage, broj pobjeda, poraza i neriješenih igara koje su se dogodile igranjem određenog poteza. S obzirom na to da je aplikacija temeljena na MCTS algoritmu, nije moguće na jednostavan način dobiti kvalitetnije objašnjenje o kvaliteti poteza osim prikazane statistike. Ako bi se aplikacija temeljila na *minimax* algoritmu, bilo bi moguće na jednostavan način opisati u koliko poteza nastaje siguran poraz odnosno pobjeda, za svaki pojedini potez. Pretraživanje cijelog prostora stanja nije moguće izvršiti u kratkom vremenu da bi iskustvo igranja igre bilo zadovoljavajuće. Stoga, kao moguće poboljšanje predlaže se uporaba *minimax* algoritma u pozadini, samo dok traje izvođenje MCTS algoritma. Tako je moguće dobiti i informacije kao što je minimalni broj poteza do sigurne pobjede ili poraza za svaki od mogućih poteza.

Primjer mogućeg vizualnog poboljšanja je isticanje četiri tokena u nizu koja čine pobjednika. Trenutno, kada korisnik aplikacije ili računalni igrač postave četiri tokena u nizu, na zaslonu se pojavljuje dijalog koji donosi informaciju o pobjedniku te nudi ponovno igranje, povratak u glavni izbornik i pregled igre. Potencijalno, bolje iskustvo bilo bi prvo istaknuti tokene kojima je jedan od igrača pobijedio, a tek nakon određenog vremena prikazati opisani dijalog. Ovakvo rješenje pak usporava ponovni početak

igre, stoga se postavlja pitanje kako bi ovakav proces utjecao na sveopće korisničko iskustvo.

Osim zvučnog efekta koji se izvodi pri padu tokena na svoje mjesto prilikom poteza jednog od igrača, ne postoji pozadinska glazba u aplikaciji. Pozadinska glazba svakako je dobrodošla uz postavke koje bi omogućavale njeno uključivanje i isključivanje.

S obzirom na to da je cilj aplikacije učenje igranja društvene igre „4 u nizu", aplikacija omogućava korisniku da zatraži neograničem broj prijedloga za poteze. Neograničeni broj prijedloga je potencijalni dvosjekli mač za postupak učenja. S jedne strane, korisnik će uvijek dobiti pomoć kada mu je potrebna, dok će s druge strane potencijalno previše koristiti pomoć, a premalo razmišljati sam. Kao potencijalno poboljšanje aplikacije se predlaže dozvoliti korisniku uporabu pomoći ograničeni broj puta ili u određenim situacijama. U tom je slučaju potrebno ispitati utjecaj na korisnikov proces učenja.

Moguće poboljšanje aplikacije je i dodavanje zaslona koji bi prikazivao razne strategije koje korisnik može koristiti u igri. Jedna takva strategija, nepar-par, spomenuta je u poglavlju 13. Strategija nepar-par ima veliki značaj, ali nije lako uočljiva igranjem protiv računalnog igrača, bez obzira na to što je računalo sklono pobjeđivati koristeći ovu strategiju. Dodavanje zaslona s objašnjenjem raznih strategija omogućilo bi korisniku bolje razumijevanje igre i bolje igranje.

13. Rezultati i rasprava

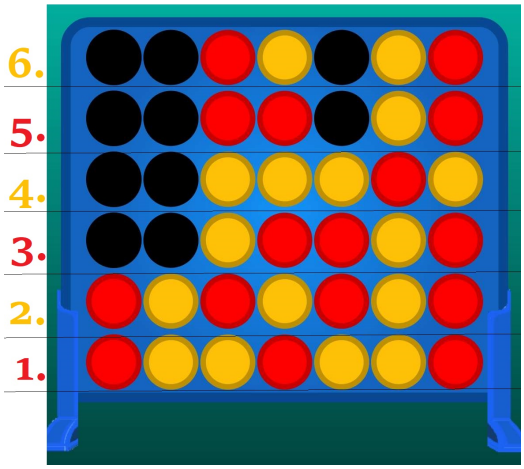
Aplikacija za igranje i učenje igranja društvene igre „4 u nizu" sposobna je igrača naučiti kvalitetnom igranju ove igre.

Ranije su navedene sve značajke koje igre pruža kako bi se korisniku omogućilo učenje igranja igre. Najznačanije od njih su mogućnost pregledavanja povijesti, izmjena poteza u bilo kojem trenutku i dobivanje preporuka za poteze. Tako je korisnik mogao vidjeti kako bi se igra odvijala da je napravio drukčiji potez, a time i naučiti iz svojih grešaka. Dovoljnom upornošću i vježbom korisnik može postati vrlo dobar u ovoj igri.

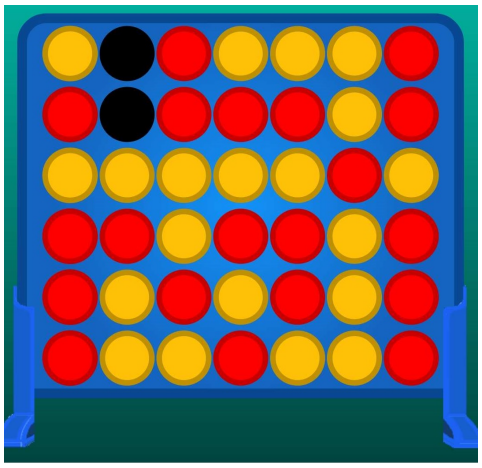
Pomoću igranja protiv računala podešenog na veće razine težine korisnik može prepoznati različite strategije koje računalo koristi, no najzanimljivija i ujedno najbolja strategija je ona pomoću koje igrač može pobijediti u zadnjem stupcu. Takva strategija se u igri „4 u nizu" naziva nepar-par (engl. *odd-even strategy*).

U strategiji nepar-par, prvog igrača nazivamo neparnim igračem zato što je on igrač koji započinje igru, dok drugog igrača nazivamo parnim igračem. Ovaj naziv proizlazli iz činjenice da neparni igrač igra neparne poteze po redu, dok parni igrač igra parne po redu, što je u kasnijem trenutku igre vrlo bitno. Iz perspektive parnog igrača, cilj igre mu je postaviti tri tokena u nizu tako da pobjedu ostvari u kasnijoj fazi, igrajući potez u zadnjem slobodnom stupcu u parnom retku. Pobjeda mu se na taj način garantira jer će neparni igrač biti prisiljen postaviti svoj token u neparni stupac. To također znači da će parni igrač svoj token postaviti na poziciju koja mu donosi pobjedu. Analogno tome, iz perspektive neparnog igrača, cilj igre mu je postaviti tri tokena u nizu tako da pobjedu ostvari u zadnjem slobodnom stupcu u neparnom retku. Slika 13.1 prikazuje kako izgleda igra u kojoj parni igrač pobjeđuje koristeći strategiju nepar-par, a ishod igre je prikazan na slici 13.2.

Iako aplikacija korisnika uči igranju igre, u igri ne postoji prikaz i objašnjenje raznih strategija. Ova strategija može se uočiti igranjem protiv računala, no korisniku nije objašnjena izravno.



Slika 13.1: Prikaz igre u kojoj parni igrač pobjeđuje koristeći strategiju nepar-par



Slika 13.2: Ishod igre u kojoj parni igrač pobjeđuje koristeći strategiju nepar-par

14. Zaključak

U sklopu ovog diplomskog rada razvijena je aplikacija za igranje i učenje igranja društvene igre „4 u nizu" (engl. *„Connect 4"*). Proučeni su razni algoritmi i tehnike koje se koriste u simulacijama društvenih igara, a posebno je istražen Monte Carlo Tree Search (MCTS) algoritam. MCTS algoritam pokazao se vrlo uspješanim u igranju raznih tipova igara, stoga je odabran kao algoritam koji se koristi u aplikaciji za igranje igre „4 u nizu". Aplikacija je implementirana tako da se ovaj algoritam koristi za određivanje poteza računalnog igrača te predlaganje poteza korisniku. Korisniku je omogućeno igranje protiv računalnog igrača na različitim razinama težine, dok se uporabom adaptivne težine igrača omogućuje korisniku postupno učenje. Nadalje, korisniku je pružena mogućnost pregledavanja povijesti igara, analiziranje poteza i učenje iz vlastitih grešaka.

U ovom radu, osim što je naveden značaj društvenih igara, opisana je i arhitektura aplikacije, korištene biblioteke, odabrana platforma za implementaciju te moguće nadogradnje i poboljšanja aplikacije. Opisana je i usporedba MCTS algoritma s drugim algoritmima te su navedeni rezultati i rasprava o mogućnostima učenja i razvoja strategija u igri „4 u nizu".

Ovaj rad spoj je tradicionalnog igranja društvenih igara s modernim pristupom učenju i razvoju strategija. Aplikacija pokazuje kako se moderni algoritmi mogu koristiti za učenje i razvoj strategija u društvenim igrama, dok korisnicima pruža zabavno i obrazovno iskustvo.

LITERATURA

[1] G. Chaslot, S. Bakkes, I. Szita, i P. Spronck. Monte-carlo tree search: A new framework for game ai. *Proceedings of the BNAIC08*, stranice 389–390, Listopad 2008. URL https://ris.utwente.nl/ws/portalfiles/portal/5100479/bnaic2008-proceedings.pdf.

[2] Jean François Dartigues. Playing board games, cognitive decline and dementia: a french population-based cohort study, Kolovoz 2013. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3758967/. Pristupljeno: 13. 4. 2024.

[3] Android Developers. Coroutines, . URL https://developer.android.com/kotlin/coroutines. Pristupljeno: 5. 6. 2024.

[4] Android Developers. Data binding library, . URL https://developer.android.com/topic/libraries/data-binding. Pristupljeno: 4. 6. 2024.

[5] Android Developers. Lifecycle, . URL https://developer.android.com/jetpack/androidx/releases/lifecycle. Pristupljeno: 3. 6. 2024.

[6] Android Developers. Livedata, . URL https://developer.android.com/topic/libraries/architecture/livedata. Pristupljeno: 3. 6. 2024.

[7] Android Developers. Navigation, . URL https://developer.android.com/guide/navigation. Pristupljeno: 4. 6. 2024.

[8] Android Developers. Room, . URL https://developer.android.com/jetpack/androidx/releases/room. Pristupljeno: 3. 6. 2024.

[9] Android Developers. View binding, . URL https://developer.android.com/topic/libraries/view-binding. Pristupljeno: 4. 6. 2024.

[10] Christy Edwards. Wellness wednesday – 5 mental benefits of board games. URL https://www.ashevillenc.gov/news/wellness-wednesday-5-mental-benefits-of-board-games/. Pristupljeno: 13. 4. 2024.

[11] Frederic Lardinois. Kotlin is now google's preferred language for android app development. URL https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/. Pristupljeno: 1. 6. 2024.

[12] Gil Press. How many people own smartphones? (2024-2029). URL https://whatsthebigdata.com/smartphone-stats/. Pristupljeno: 1. 6. 2024.

[13] Android Open Source Project. App sandbox, . URL https://source.android.com/docs/security/app-sandbox. Pristupljeno: 1. 6. 2024.

[14] Android Open Source Project. Security features, . URL https://source.android.com/docs/security/features. Pristupljeno: 1. 6. 2024.

[15] Sphinx Solution. How much does it cost to put an app on the app store? URL https://www.sphinx-solution.com/blog/cost-to-put-an-app-on-the-app-store/. Pristupljeno: 1. 6. 2024.

[16] StatCounter Global Stats. Mobile operating system market share worldwide. URL https://gs.statcounter.com/os-market-share/mobile/worldwide. Pristupljeno: 1. 6. 2024.

[17] Wikipedia. Android (operating system) - history. URL https://en.wikipedia.org/wiki/Android_(operating_system)#History. Pristupljeno: 1. 6. 2024.

[18] Mark H. M. Winands. *Monte-Carlo Tree Search*, stranice 1179–1184. Springer International Publishing, Cham, 2024. ISBN 978-3-031-23161-2. doi: 10.1007/978-3-031-23161-2_12. URL https://doi.org/10.1007/978-3-031-23161-2_12.

POPIS SLIKA

Implementacija MCTS algoritma u obrazovnoj aplikaciji za igranje društvene igre "4 u nizu"

Sažetak

U sklopu ovog rada razvijena je aplikacija za igranje i učenje igranja društvene igre „4 u nizu" (engl. *„Connect 4"*). Aplikacija koristi Monte Carlo Tree Search (MCTS) algoritam za određivanje kvalitetnih poteza računalnog protivnika, kao i za preporučivanje poteza igraču. Unutar ovog rada opisano je sve što je potrebno za razumijevanje MCTS algoritma i njegove implementacije u aplikaciji koja pruža moderan pristup učenju.

Ključne riječi: MCTS, igra, aplikacija, društvene igre, „4 u nizu", učenje, strategija, igrifikacija, obrazovanje

Implementation of the Monte Carlo Tree Search Algorithm in an Educational Application for Playing the Board Game "Connect Four."

Abstract

In this thesis, an application for playing and learning to play the board game "Connect Four" has been developed. The application uses the Monte Carlo Tree Search (MCTS) algorithm to determine the quality of the moves of the computer opponent, as well as to recommend moves to the player. This thesis describes everything needed to understand the MCTS algorithm and its implementation in an application that provides a modern approach to learning.

Keywords: MCTS, game, application, board games, "Connect Four", learning, strategy, gamification, education