Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Departman za matematiku i informatiku
Odsek za informatiku

Matija Kljajić 137/22

Implementacija igre sa nultom sumom i minimaks algoritma u Pharo okruženju

Seminarski rad B - Objektno programiranje uživo uz Pharo

Novi Sad
2024

Apstrakt

Kontekst: Pharo pruža fleksibilnu platformu za posmatranje i dalji razvitak softverskih rešenja, dok je rešavanje igara sa nultom sumom dobar primer za prikaz fundamentalnih koncepata veštačke inteligencije u ovom okruženju.
Cilj: Namera ovog projekta jeste da se dizajnira i implementira klasična verzija igre „iks-oks“ i pored toga implementira veštačka inteligencija za rešavanje ove igre u Pharo okruženju.
Metoda: Primenom Pharo okruženja će se realizovati model igre, a izgradnja grafičkog korisničkog interfejsa će se olakšati pomoću Bloc razvojnog okvira. Potezi računara će biti simulirani korišćenjem osnovnog minimaks algoritma.
Ishod: Kako je reč o jednostavnoj igri sa nultom sumom gde primenjujemo minimaks algoritam, simulacija igre postaje „nepobediva“ u slučaju da se partija igre igra protiv računara.
Zaključak: Rad pokazuje primenu minimaks algoritma u igri „iks-oks“, što može poslužiti kao temelj za realizaciju drugih igara sa nultom sumom u Pharo okruženju.

Ključne reči: Pharo, Bloc, veštačka inteligencija, minimax

---

1. Uvod

Domenu teorije igara pripadaju igre sa nultom sumom u kojima je dobitak jednog igrača tačno kompenzovan gubitkom drugog, čineći zbir svih dobitaka i gubitaka uvek ekvivalentnim nuli. One omogućavaju preciznu analizu i modeliranje strategija igrača, pružajući uvid u optimalne odluke koje igrači mogu doneti u različitim situacijama. Iz tog razloga su idealne za primenu i testiranje algoritama veštačke inteligencije, koji imaju za cilj da pronađu optimalnu strategiju za svakog igrača.

Jedan od najjednostavnijih primera igre sa nultom sumom jeste klasična igra po imenu „iks-oks“. Ova igra, iako jednostavna, pruža odličnu osnovu za pregled i razumevanje već spomenutih algoritama. Kroz ovaj rad ćemo pokušati da je realizujemo uz minimaks algoritam koji rekurzivnom pretragom prolazi kroz sve moguće scenarije i kao rezultat vraća strategiju koja minimizuje mogući gubitak. (Russell and Norvig, 2010)

Pharo je okruženje otvorenog koda i njegova glavna karakteristika jeste manipulacija objektima u realnom vremenu. Zbog toga, Pharo omogućava prilagodljivost koja ga čini idealnim za postepen i siguran, ali istovremeno solidno brz razvoj softverskih rešenja. Kako vuče korene iz Smalltalk programskog jezika, Pharo nasleđuje njegovu vrlo prostu sintaksu, koja je poznata po svojoj jednostavnosti i eleganciji. Ova karakteristika omogućava da se cela sintaksa jezika može opisati na vrlo sažet način, što je osobina koju je Ralf Džonson, jedan od koautora knjige Design Patterns iz 1994. godine, slikovito opisao tvrdeći da se sintaksa Smalltalk-a može smestiti na jednu razglednicu. (Ducasse, 2000)

Još jedna snaga Pharo ekosistema jeste Bloc, razvojni okvir niskog nivoa koji nam olakšava razvoj grafičkih korisničkih interfejsa što je ključno za izradu interaktivnih video-igara. On pruža upravljanje grafičkim elementima, događajima i animacijama, omogućavajući nam da se fokusiramo više na samu logiku programa i algoritam. (Ducasse, Demeulenaere and Dias, 2024) Dakle, u slučaju implementacije igre „iks-oks“, Pharo nudi sve neophodne alate za razvoj, testiranje i vizuelizaciju rezultata kroz interaktivni interfejs, čime omogućava jednostavan i efikasan razvoj složenih algoritamskih rešenja.

2. Organizacija

Projekat je organizovan u tri ključna paketa: *OXO*, *OXO-AI*, i *OXO-Tests*. Ovakva organizacija omogućava jasnu separaciju klasa za logiku igre, veštačku inteligenciju i testiranje. Ovakav primer strukture doprinosi lakšem održavanju i proširenju programa. Paket *OXO* služi kao osnovni deo aplikacije uz pomoć obeležja (eng. *tag*) *Model*, *View*, i *Game*. U ovom paketu, *Model* sadrži osnovnu logiku igre, dok *View* upravlja grafičkim korisničkim interfejsom koristeći Bloc razvojni okvir. *Game* se sastoji od klase koja predstavlja inicijalizator naše igre. Ne pratimo potpun *model-view-controller* (MVC) projektni obrazac zato što u ovom slučaju zanemarujemo *controller* modul kako nam Bloc razvojni okvir pojednostavljuje upravljanje događajima dok sam projekat nema dovoljno veliki stepen kompleksnosti da bi mogli da opravdamo takvu vrstu arhitekture.

3. Izrada modela

Izrada modela međutim predstavlja prvi korak u procesu implementacije igre. Model igre definiše osnovne strukture i pravila, pružajući osnov na kojem će se dalje graditi logika igre. Kreiraćemo modularan i fleksibilan model koji će se lako proširivati i prilagođavati za dalje iteracije i optimizacije u slučaju da su potrebne. Model će biti prilagođen za integraciju sa minimaks algoritmom koji će se kasnije implementirati. Sve klase vezane za model se nalaze u Model obeležju unutar *OXO* paketa. Cilj projekta jeste simulacija „iks-oks“ igre tako da ako pogledamo sliku 1 na kojoj se može videti primer partije ove igre, možemo zaključiti da nam je potrebna tabla popunjena ćelijama koje možemo popuniti „X“ i „O“ simbolima. To možemo realizovati implementacijom *OXOGrid* i *OXOElement* klasa koje će respektivno predstavljati tabelu i ćeliju igre.

3.1. Izrada testiranjem

Testiranje implementacije osigurava da sve komponente sistema funkcionišu ispravno i u skladu sa našim zahtevima. U *OXO-Tests* paketu, za svaku klasu modela razvijamo odgovarajuću test klasu koja nosi u imenu sufiks *Test* i koja nasleđuje klasu *TestCase*. Na osnovu njih realizujemo sve klase u paketu *OXO*. Ovakav pristup, poznat kao razvoj vođen testovima (eng. *Test Driven Development*), nam omogućava da prvo definišemo očekivano ponašanje sistema kroz testove, što olakšava samu implementaciju našeg modela. (Ducasse et al., 2022)

3.2. OXOCell

Želimo da *OXOCell* bude klasa koja pravilno upravlja stanjem ćelije, osiguravajući da se jednom upisano stanje ne može promeniti. Testiranjem ćemo proveriti da li *OXOCell* inicijalno ima ispravno stanje, kao i da li *setter* metoda pravilno ograničava izmenu stanja nakon što je ćelija popunjena.

OXOCell je jednostavna klasa koja sadrži varijablu instance *state*, i ona služi kao skladište za vrednost upisanu u ćeliju. Imamo osnovne *getter* i *setter* metode za upravljanje *state* promenljivom, pri čemu je u *setter* metodi implementirana restrikcija koja sprečava promenu stanja nakon što je ćelija već popunjena. To se jasno može videti u kodu metode prikazanom ispod u listingu 1.

```
state: aState

    ^ state
          ifNil: [ state := aState. true ]
          ifNotNil: [ false ]
```

Listing 1 – Setter metoda za state

Radi lakšeg testiranja i čitavog prikaza modela, uređujemo metodu *printOn* za *OXOCell*. Ona prilagođava kako će naš objekat biti prikazan u tekstualnom formatu. Može se videti kako je implementirana metoda kroz listing 2.

```
printOn: aStream

    state
         ifNil: [ aStream nextPutAll: '' ]
         ifNotNil: [ aStream nextPutAll: state ]
```

Listing 2 – Metoda printOn

3.3. OXOGrid

Želimo postići da klasa *OXOGrid* pravilno predstavlja i upravlja tabelom koja se koristi za igru. Ona treba da pravilno beleži broj poteza, prati koji je igrač na redu i identifikuje pobednika ili nerešen ishod kada je to traženo. Testiranjem ćemo proveriti da li *OXOGrid* inicijalno ispravno pravi tabelu u koju će se upisivati simboli „X“ i „O“, da li pravilno broji poteze i da li tačno određuje ko je na redu i kako je igra završena.

Klasa *OXOGrid* obuhvata sva pravila koja definišu model naše igre. Sadrži varijable instance *cells*, *moves*, *turn*, i *winner*. Prilikom inicijalizacije, varijabla *cells* se postavlja kao dvodimenzionalni niz sa 3 kolone i 3 reda, koji se zatim popunjava uz pomoć metode *fillWithCells*. Varijabla *moves*, koja prati broj napravljenih poteza, inicijalizuje se na nulu. Varijable *turn* i *winner* ostaju prazne.

```
initialize

    cells := Array2D new: 3.
    self fillWithCells.
    moves := 0
```

Listing 3 – Inicijalizator OXOGrid objekta

Metoda *fillWithCells* popunjava varijablu *cells* kreiranjem novih objekata tipa *OXOCell* za svaku ćeliju naše tabele.

```
fillWithCells

     cells indicesDo: [ :row :col |
          cells at: row at: col put: (OXOCell new) ]
```

Listing 4 – Pomoćna metoda fillWithCells

Metodom *turn* vraćamo simbol koji treba sledeće da se upiše na osnovu broja odigranih poteza i tako pratimo koji igrač jeste na redu.

```
turn

     turn := moves % 2 = 0
                  ifTrue: [ 'O' ]
                  ifFalse: [ 'X' ].
     ^ turn
```

Listing 5 – Metoda turn koja vraća igrača koji je na redu

Potez u modelu se pravi metodom *makeMove*, kojoj se prosleđuje objekat tipa *OXOCell* na kome se vrši potez. Ova metoda popunjava datu ćeliju i, u slučaju uspešnog poteza, pomoću metode *addMove* povećava vrednost varijable *moves* za jedan.

```
makeMove: aCell
     "Make move at said row and at said column if it's empty."

     aCell ifNotNil:
          [ (aCell state: self turn)
               ifTrue: [ self addMove ] ]
```

Listing 6 – Metoda makeMove

Metoda *isOver* proverava da li je igra završena. Ova metoda koristi pomoćne metode *checkInCol*, *checkInRow*, *checkInDiagonals* i *checkSet* da bi utvrdila da li ima pobednika. Metoda funkcioniše tako što najpre proverava da li je napravljen dovoljan broj poteza, a zatim poziva pomoćne metode radi provere. Na kraju, na osnovu vrednosti varijable *winner*, vraća informaciju o tome da li je igra završena ili ne. Pomoćne metode proveravaju sve moguće kombinacije na tabeli i oslanjaju se na metodu *checkSet*, koja koristi ugrađene kolekcije za skupove u Pharo okruženju kako bi utvrdila da li postoji pobednik.

```
isOver

     moves >= 5 ifFalse: [ ^ false ].
     1 to: 3 do: [ :idx |
          self checkInRow: idx.
          self checkInCol: idx ].
     self checkInDiagonals.

     (moves == 9 or: winner isNotNil)
          ifTrue: [ ^ true ]
          ifFalse: [ ^ false ]
```

Listing 7 – Metoda isOver koja vrši proveru da li je igra gotova

Metoda *checkInRow* proverava da li postoji pobednik u zadatom redu.

```
checkInRow: idx

     | set |
     winner ifNil: [
          set := ((self grid atRow: idx)
               collect: [ :x | x state ]) asSet.
          self checkSet: set ].
     ^ winner
```

Listing 8 – Pomoćna metoda checkInRow

Metoda *checkInCol* proverava kolone na isti način.

```
checkInCol: idx

     | set |
     winner ifNil: [
          set := ((self grid atColumn: idx)
               collect: [ :x | x state ]) asSet.
          self checkSet: set ].
     ^ winner
```

Listing 9 – Pomoćna metoda checkInCol

Metoda *checkInDiagonals* proverava obe dijagonale.

```
checkInDiagonals

     | set1 set2 |
     winner ifNil: [
          set1 := (self grid diagonal
                    collect: [ :x | x state ]) asSet.
          set2 := Set
               newFrom: { (self grid at: 1 at: 3) state.
                        (self grid at: 2 at: 2) state.
                        (self grid at: 3 at: 1) state }
               asOrderedCollection.
          self checkSet: set1.
          self checkSet: set2 ].
     ^ winner
```

Listing 10 – Pomoćna metoda checkInDiagonals

Konačno, metoda *checkSet* proverava da li su svi simboli u zadatom skupu isti, čime identifikujemo potencijalnog pobednika.

```
checkSet: aSet

     aSet size == 1 ifFalse: [ ^ self ].
     (aSet includes: 'X') ifTrue: [ winner := 'X' ].
     (aSet includes: 'O') ifTrue: [ winner := 'O' ]
```

Listing 11 – Pomoćna metoda checkSet

4. Izrada grafičkog korisničkog interfejsa

Korišćenjem Bloc razvojnog okvira, dizajniramo sve potrebne elemente grafičkog korisničkog interfejsa. Bloc omogućava kreiranje grafičkih elemenata putem objekata koji nasleđuju klasu *BlElement*, dok obrada događaja, poput klika miša, postaje jednostavna uz korišćenje metode *addEventHandlerOn* i objekta *BlClickEvent*.

4.1. OXOCellElement

OXOCellElement je element koji predstavlja ćeliju igre unutar grafičkog korisničkog interfejsa i njegov izgled konfigurišemo pri inicijalizaciji. Pomoćna metoda *initializeText* inicijalizuje objekat klase *BlTextElement*, koji služi za prikaz stanja *OXOCell* objekta unutar ćelije, što se realizuje kroz metodu *fillCell*. Ova metoda omogućava prikaz stanja ćelije uz odgovarajuću vizuelnu reprezentaciju. Takođe ovde omogućavamo obradu događaja klikom na ćeliju.

```
initialize
     super initialize.
     self initializeText.

     self size: 68 @ 68.
     self layout: BlLinearLayout new alignCenter.
     self background: (BlColorPaint color: Color black).

     self cell: OXOCell new.

     self addEventHandlerOn: BlClickEvent do:
          [ :anEvent | self click ].
```

Listing 12 – Inicijalizator OXOCellElement objekta

```
fillCell: aState

     text text:
          (self cell state asRopedText fontSize: 64;
          foreground: Color white).
     self addChild: text
```

Listing 13 – Metoda fillCell

Metoda *click*, koja je pozvana prilikom klika na ćeliju kako je definisano pri inicijalizaciji, prvo proverava da li je potrebno resetovati igru, što se radi metodom *reset*, objašnjenom u poglavlju o pokretanju igre. Ukoliko je kliknuta već popunjena ćelija, generiše se upozorenje, a ako nije, igra se nastavlja sa potezima igrača i računara preko metoda *makeMove* i *makeAIMove*.

```
click

     (self parent model moves = 9 or: self parent model winner
isNotNil)
          ifTrue: [ ^ self parent game reset ].

     self cell state
          ifNotNil:
               [ UIManager default inform:
                    'Tried to edit already filled cell.'.
                    42.
                    ^ self ].

     self makeMove.
     self checkWinCondition.

     self parent model winner ifNil: [
          self makeAIMove.
          self checkWinCondition ]
```

Listing 14 – Metoda click koja se izvršava pri događaju

```
makeMove

     self parent model makeMove: self cell.
     self fillCell: self parent model turn
```

Listing 15 – Metoda makeMove

U slučaju da veštačka inteligencija nije uključena, metoda *makeAIMove* se zaobilazi. Kasnije ćemo izvršiti implementaciju *OXOAI* klase i *makeBestMove* metode na koju će se *makeAIMove* metoda oslanjati.

```
makeAIMove

     | pickedCell |
     self parent ai ifFalse: [ ^ self ].
     pickedCell :=
          (OXOAI new: self parent model) makeBestMove.
     self parent model makeMove: pickedCell.
     self parent childrenDo: [ :cellEl |
          cellEl cell = pickedCell ifTrue:
               [ cellEl fillCell: self parent model turn ] ].
```

Listing 16 – Metoda makeAIMove

Nakon svakog poteza, vrši se provera stanja igre da li je došlo do nerešenog rezultata ili pobede preko metode ispod. Ako se ispostavi da je igra završena, sve ćelije se boje odgovarajućom bojom koristeći metodu *changeChildrenColor* koja se može naći u *OXOGridElement.*

```
checkWinCondition

     (self parent model isOver
          and: self parent model winner = 'O')
          ifTrue: [ self parent
               changeChildrenColor: Color green twiceDarker ]
          ifFalse: [
               self parent model winner = 'X' ifTrue: [
                    self parent changeChildrenColor:
                         Color red twiceDarker ].
               self parent model moves = 9 ifTrue: [
                    self parent changeChildrenColor:
                         Color yellow twiceDarker ] ]
```

Listing 17 – Metoda checkWinCondition

4.2. OXOGridElement

Za samu tabelu takođe inicijalizacijom konfigurišemo izgled. *OXOGridElement* ima za varijable instance varijable *game*, *model* i *ai*. Varijablu *ai* koristimo za proveru da li imamo računar kao protivnika, međutim pri osnovnom instanciranju ovog objekta stavljamo *ai* da bude isključen.

```
initialize

     super initialize.
     self background: Color white.
     self layout: (BlGridLayout horizontal cellSpacing: 5).
     self layout columnCount: 3.

     self constraintsDo: [ :layoutConstraints |
          layoutConstraints horizontal fitContent.
          layoutConstraints vertical fitContent ].

     self model: OXOGrid new.
     self ai: false
```

Listing 18 – Inicijalizator OXOGridElement objekta

Sve *getter* i *setter* metode su trivijalne osim *model setter* metode koja postavlja *OXOCellElement* za svaki *OXOCell* iz *OXOGrid* objekta koji predstavlja naš model.

```
model: anOXOGrid

     model := anOXOGrid.
     model grid do: [ :anOXOCell |
          self addChild: (OXOCellElement new: anOXOCell) ]
```

Listing 19 – Setter metoda za model

Takođe ovde imamo *changeChildrenColor* metodu koju smo već pomenuli i koja menja početnu konfiguraciju boje za svaki *OXOCellElement*.

```
changeChildrenColor: aColor

	self childrenDo: [ :x | x background: aColor ]
```

Listing 20 – Metoda changeChildrenColor

Da bi napravili razliku između *OXOGridElement* sa uključenom i isključenom veštačkom inteligencijom, konstruišemo dve metode na nivou klase.

```
new: aBlSpace

	^ self new game: aBlSpace
```

Listing 21 – Klasna metoda za kreiranje OXOGridElement objekta

```
newWithAI: aBlSpace

	^ (self new game: aBlSpace) ai: true
```

Listing 22 – Klasna metoda za kreiranje uz uključenu veštačku inteligenciju

5. Pokretanje igre

Game obeležje u *OXO* paketu sadrži *OXO* klasu sa varijablama instance *game* i *space*. Ona sopstvenim instanciranjem instancira *BlSpace* objekat iz Bloc razvojnog okvira pod njenom space varijablom. *BlSpace* objekat predstavlja novi sistemski prozor na koji možemo da dodajemo *BlElement* objekte. (Ducasse, Demeulenaere and Dias, 2024) Instanciramo *OXO* preko njenih metoda na nivou klase *playVsPlayer* ili *playVsComputer* koje dodaju *BlSpace* elementu adekvatan *OXOGridElement* objekat inicijalizovan preko *new* ili preko *newWithAi* metode koji se realizuje u *View* obeležju. Na listingu ispod možemo videti metodu *playVsPlayer* metodu koja će instancirati naš progam.

```
playVsPlayer

	| newGame |

	newGame := self new.

	newGame game: (OXOGridElement new: newGame).
	newGame space root addChild: newGame game.
	newGame space root whenLayoutedDoOnce:
		[ newGame space extent: newGame game size ].

	newGame space show.

	^ newGame
```

Listing 23 – Pokretač igre bez veštačke inteligencije

Ovim vraćamo naš objekat kojim prikazujemo *BlSpace* element, a igru resetujemo preko sledeće metode. Ne menjamo *space* varijablu, već samo inicijalizujemo novi game na osnovu toga koji je tip igre pokrenut.

```
reset

    game := (self space root childAt: 1) ai
                ifTrue: [ OXOGridElement newWithAI: self ]
                ifFalse: [ OXOGridElement new: self ].

    space root removeChildren.
    space root addChild: game
```

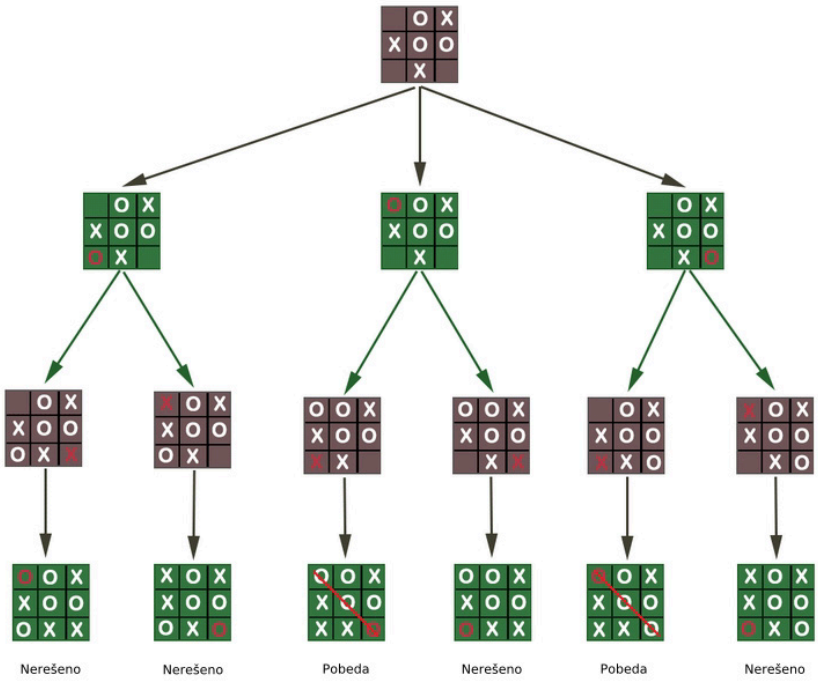
Listing 24 – Metoda za resetovanje partije igre

Za sada imamo neku osnovnu vrstu igre koju mogu igrati dva ljudska igrača, ali koju možemo unaprediti implementacijom algoritma koji će nam omogućiti *makeAIMove* metodu.

6. Izrada minimaksa

6.1. Objašnjenje algoritma

Minimaks algoritam jeste rekurzivan algoritam za odabir sledećeg najboljeg poteza u igrama sa n brojem igrača. Oslanjamo se na generisanje stabla koje pokriva sve moguće scenarije u igri. Algoritam će biti opisan u kontekstu „iks-oks“ igre. Koren stabla jeste trenutna situacija na tabli, dok je svaki susedni čvor sledeći mogući scenario koji može biti odigran. Svaki nivo u stablu predstavlja sve moguće poteze od igrača koji je u tom trenutku na redu. Za svaki čvor se može izračunati težinska vrednost koja bi predstavljala koji igrač ima prednost. (Russell and Norvig, 2010) Na primer, ako imamo igrače A i B, scenario gde igrač A pobeđuje vraća pozitivnu vrednost, dok scenario gde igrač A gubi vraća negativnu vrednost. Na osnovu toga možemo da zaključimo najbolji mogući potez prolaskom kroz stablo.



Slika 1 – Primer stabla igre

6.2. Implementacija veštačke inteligencije

Pre implementacije minimaksa se mora prvo izvršiti implementacija metoda koje će nam resetovati stanja prilikom rekurzije koju budemo izvršavali kako ne bi poremetili model. Potrebne su nam metode *undoMove* koja će pripadati klasi *OXOGrid* i *reset* koja će pripadati klasi *OXOCell*.

```
undoMove: aCell

	aCell reset.
	self undoMove.
```

Listing 25 – Metoda undoMove

```
undoMove

	moves := moves - 1
```

Listing 26 – Pomoćna metoda undoMove

```
reset

	state := nil
```

Listing 27 – Metoda za resetovanje stanja ćelije

U paketu *OXO-AI* pravimo *OXOAI* klasu u kojoj će se realizovati minimaks algoritam i koja ima varijablu instance *board*. Pri instanciranju će se proslediti trenutno popunjen model igre u tu varijablu. Implementiramo metodu *makeBestMove* koja će pokrenutni *minimax* metodu za svaku ćeliju u našem modelu i na osnovu toga će se izvući ćelija sa najvećom težinskom vrednošću.

```
makeBestMove

	| bestVal bestMove |
	bestVal := SmallInteger minVal.
	bestMove := nil.

	self board grid indicesDo: [ :row :col |
		| cell |
		cell := self board grid at: row at: col.
		cell state ifNil: [
			| moveVal |
			self board makeMove: cell.
			moveVal := self minimax: 0 isMax: false.
			self board undoMove: cell.
			moveVal > bestVal ifTrue: [
				bestMove := cell.
				bestVal := moveVal ] ] ].
	^ bestMove
```

Listing 28 – Metoda makeBestMove

Pozivanjem metode *minimax* izračunavamo težinsku vrednost preko *evaluate* metode i konfigurišemo težinsku vrednost dubinom pretrage kako bi naša veštačka inteligencija brže pobeđivala, a sporije gubila. Na osnovu prvog parametra metode *minimax* beležimo vrednost dubine pretrage, a drugim parametrom proveravamo da li se rezultat minimizira ili maksimizira to jest za kog igrača računamo finalno stanje. Metodama *min* i *max* se upravo to radi i sa njima rekurzivno zalazimo dublje kroz svaki mogući scenario dok nam *minimax* ne vrati vrednost koja će označavati pobedu, gubitak ili nerešeno stanje.

```
minimax: depthNum isMax: maxxer

     | score |
     score := self evaluate.

     (score = -10) ifTrue: [ ^ score + depthNum ].
     (score = 10) ifTrue: [ ^ score - depthNum ].
     self movesLeft ifFalse: [ ^ 0 ].

     maxxer
          ifTrue: [ ^ self max: depthNum + 1 isMax: maxxer ]
          ifFalse: [ ^ self min: depthNum + 1 isMax: maxxer ]
```

Listing 29 – Glavna minimax metoda

```
min: depthNum isMax: maxxer

     | best |
     best := SmallInteger maxVal.
     self board grid indicesDo: [ :row :col |
          | cell |
          cell := self board grid at: row at: col.
          cell state ifNil: [
               self board makeMove: cell.
               best := best min:
                    (self minimax: depthNum isMax: maxxer not).
               self board undoMove: cell ] ].
     ^ best
```

Listing 30 – Minimizujuća metoda

```
max: depthNum isMax: maxxer

     | best |
     best := SmallInteger minVal.
     self board grid indicesDo: [ :row :col |
          | cell |
          cell := self board grid at: row at: col.
          cell state ifNil: [
               self board makeMove: cell.
               best := best max:
                    (self minimax: depthNum isMax: maxxer not).
               self board undoMove: cell ] ].
     ^ best
```

Listing 31 – Maksimizujuća metoda

Metoda *evaluate* vrši najbitniju funkciju kako se po njoj konfigurišu pravila naše veštačke inteligencije. Ona se oslanja na već odrađene metode u modelu koje vrše prostu proveru da li ima pobednika ili ne. Kako je igra „iks-oks“ kratka i krajnje jednostavna, ne moramo da osmišljamo drugačiju vrstu provere koja bi izvršavala restrikciju koliko može pretraga da ide u dubinu.

```
evaluate

    self board isOver.

    self board winner = 'X'
        ifTrue: [ self board winner: nil. ^ 10 ].
    self board winner = 'O'
        ifTrue: [ self board winner: nil. ^ -10 ].

    ^ 0
```

Listing 32 – Metoda za izračunavanje stanja scenarija

Ovom metodom završavamo implementaciju osnovne verzije minimaksa i osposobljavamo *makeAIMove* metodu koja vraća najoptimalniji potez. Metodom *playVsComputer* se sada može uspešno pokrenuti partija igre sa uključenom veštačkom inteligencijom.

7. Zaključak

U ovom radu smo uspešno realizovali klasičnu verziju igre "iks-oks" koristeći Pharo okruženje. Napravljen je model koji je uspešno prikazan kroz Bloc razvojni okvir. Postignuti rezultati naglašavaju snagu Pharo okruženja u razvoju algoritamskih rešenja, posebno kada su u pitanju aplikacije koje zahtevaju brze iteracije i dinamičko testiranje. Ograničenja rada uključuju jednostavnost modela i prikaza igre, ali upravo ova jednostavnost omogućava jasan prikaz minimaks algoritma. Uz optimizacije minimaks algoritma, slični principi razvoja softvera se mogu upotrebiti nad razvojem kompleksnijih igara sličnog tipa. (Russell and Norvig, 2010)

Ovaj projekat ne samo da demonstrira primenu veštačke inteligencije u jednostavnim igrama, već i ukazuje na fleksibilnost i moć Pharo okruženja kao alata za razvoj složenih algoritamskih rešenja.

8. Literatura

- Ducasse S (2000). Object-Oriented Design with Smalltalk - a Pure Object Language and its Environment. [online] Available at: https://scg.unibe.ch/archive/lectures/DucasseLectures/Duca00y1SmalltalkLectures.pdf.
- Ducasse S, Demeulenaere E and Dias M (2024). Building a memory game with Bloc. BoD - Books on Demand ed. [online] Available at: https://books.pharo.org/booklet-ASimpleMemoryGameInBloc/2024-06-05-ASimpleBlocTutorial.pdf.
- Ducasse S, Rakic G, Sebastijan Kaplar and Ducasse Q (2022). Pharo 9 by example. BoD - Books on Demand.
- Russell S and Norvig P (2010). Artificial Intelligence: A Modern Approach. 3rd ed. New Jersey: Pearson.