Implementacija igre sa nultom sumom uz minimaks algoritam u Pharo okruženju

Seminarski rad – Objektno programiranje uživo uz Pharo

Autor: *Matija Kljajić*
Mentor: *dr Gordana Rakić*

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Avgust 2024

Sažetak

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 kreira 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: Ovaj rad pokazuje uspešnu 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, Smalltalk, Bloc, Objektno orijentisano programiranje, Minimaks, Veštačka inteligencija u igrama, Testiranje, Razvoj video-igrica

---

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 nekog od 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 i Norvig 2009.).

Pharo je okruženje otvorenog kôda 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 elegantnosti. 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 pod nazivom „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 (engl. *framework*) niskog nivoa koji nam olakšava razvoj grafičkih korisničkih interfejsa (GUI-a) 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 izradu algoritama za iste (Ducasse, Demeulenaere i Dias 2024.). Dakle, u slučaju implementacije igre „iks-oks“, Pharo nudi sve neophodne alate za razvoj, testiranje i vizuelizaciju rezultata kroz interaktivni interfejs, čime se omogućava jednostavan i efikasan razvoj složenih algoritamskih rešenja.

2 Plan rada

2.1 Pregled zahteva

Cilj rada jeste implementacija igre sa nultom sumom u Pharo okruženju. Zahtevi za izradu programa sa kojim se suočavamo nisu kompleksni sami po sebi. U ovom radu će se u obzir uzeti standardna pravila igre „iks-oks“. Skalirati igru je trivijalno u slučaju da je to potrebno kako su pravila minimalna. Standardna igra uključuje dva igrača i ona se uobičajeno igra na papiru gde je nacrtana tabela koja se može predstaviti matricom 3×3. Igrači naizmenično postavljaju simbole koje su izabrali na početku igre i njihov cilj jeste da zaređaju tri ista simbola dijagonalno, uspravno ili vodoravno. Onaj koji prvi to uspe da uradi jeste pobednik. Nerešen finalni status igre je moguć i on biva najčešći rezultat ako oba igrača igraju bez greške. U teoriji igara se takva igra naziva uzaludnom igrom (eng. *futile game*) (Wang 1993.).

2.2 Organizacija projekta

Program će se podeliti na nekoliko paketa koji će se svoditi na rešavanje specifičnih komponenti celokupnog sistema programa. Za uspešan rad programa je potreban model za logiku igre, skup elemenata koji će uspešno prikazati taj model korisniku preko neke vrste GUI-a, komponenta za automatsko izvršavanje poteza (simulacija veštačke inteligencije namenjene za rešavanje igre) i komponenta za testiranje svih pomenutih delova sistema. Pharo ekosistem podržava raspodelu klasa na pakete u kojima se klase mogu grupisati uz pomoć obeležja (engl. *tag*) u slučaju veće kompleksnosti sistema. Ovakvom raspodelom programa projekat biva skalabilan, lakši za dokumentaciju i distribuciju.

Ključni paketi ovog projekta jesu OXO-Model paket koji sadrži model, OXO-View paket koji sadrži GUI elemente i OXO-Game paket koji sadrži inicijalizatore igre. Implementacija minimaks algoritma se nalazi u OXO-AI paketu, dok se svi testovi za projekat nalaze u OXO-Tests paketu. Testovi OXO-Tests paketa su grupisani pomoću obeležja na osnovu ostalih paketa. Testovi služe kao još jedan vid dokumentacije za projekat i oni smanjuju mogućnost pojave nepredviđene greške. Celokupan kôd je napisan na principu razvoja vođenom testovima (engl. *test-driven development*). Nivo interaktivnosti koji možemo imati sa Pharo okruženjem nam dopušta lako izvođenje ove metodologije za razvoj sistema (Ducasse, Rakić i Kaplar 2022.).

Arhitektura programa odrađenog kroz ovaj rad se može reći da je nalik MVC arhitekturi (engl. *model-view-controller*). Jedan od ciljeva MVC arhitekture jeste dekomponovanje programa radi postizanja fleksibilnosti kôda (Gamma i dr. 1994.). Zahvaljujući jednostavnosti zahteva željenog programa ne moramo potpuno preslikati MVC arhitekturu i realizovati ovaj projektni obrazac (engl. *design pattern*). Ispoštovaćemo komponentu modela i prikaza modela (engl. *view*), respektivno, kroz klase u već spomenutim OXO-Model i OXO-View paketima. Od korisnika se očekuje samo akcija klika mišem na odgovarajuću ćeliju u tabeli koja predstavlja igru, tako da nam u ovom slučaju nije potreban potpuno odvojen kontrolor (engl. *controller*).

3 Izrada modela

Izrada modela predstavlja prvi korak u procesu implementacije igre. Model igre definiše osnovne strukture podataka i pravila (Gamma i dr. 1994.), pružajući osnov na kojem se dalje može 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. Realizacija prikaza podataka to jest modela će se izvršiti preko posebnih OXOGrid i OXOElement klasa iz OXO-Model paketa.

3.1 OXOCell

Želimo da OXOCell bude klasa koja pravilno upravlja stanjem ćelije, osiguravajući da se jednom upisano stanje ne može promeniti.

OXOCell je jednostavna klasa koja sadrži varijablu instance state, i ona služi kao skladište za vrednost ćelije tabele. 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 kôdu metode prikazanom u listingu 1.

```
1 OXOCell >> state: aState
2     ↑ state
3     ifNil: [ state := aState. true ]
4     ifNotNil: [ false ]
```

Listing 1: OXOCell setter metoda za state

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.

Radi lakšeg kasnijeg testiranja i čitavog prikaza modela, uređujemo specijalnu metodu printOn za OXOCell. Ona prilagođava kako će naš objekat biti prikazan u tekstualnom formatu to jest pošiljaoca metode vraćamo u tekstualnom obliku (Ducasse, Rakić i Kaplar 2022.). Može se videti kako je implementirana metoda kroz listing 2.

```
1 OXOCell >> printOn: aStream
2     state
3          ifNil: [ aStream nextPutAll: '' ]
4          ifNotNil: [ aStream nextPutAll: state ]
```

Listing 2: OXOCell printOn metoda

3.2 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. (Listing 3)

```
1 OXOGrid >> initialize
2     cells := Array2D new: 3.
3     self fillWithCells.
4     moves := 0
```

Listing 3: Inicijalizator OXOGrid objekta

Metoda fillWithCells popunjava varijablu cells kreiranjem novih objekata tipa OXOCell za svaku ćeliju naše tabele. (Listing 4)

```
OXOGrid >> 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 što se tiče poteza. Kako u originalnim pravilima nije definisano, pretpostavljamo da igrač koji igra simbolom „O“ igra prvi. (Listing 5)

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

Listing 5: Metoda turn koja vraća igrača čiji je potez

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. (Listing 6)

```
OXOGrid >> makeMove: aCell
    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. (Listing 7)

```
OXOGrid >> 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 što se može videti u listingu 8.

```
OXOGrid >> 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. (Listing 9)

```
OXOGrid >> 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. (Listing 10)

```
OXOGrid >> 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. (Listing 11)

```
OXOGrid >> 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 Grafički korisnički interfejs

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. (Listing 12) Pomoćna metoda initializeText inicijalizuje objekat klase BlTextElement, koji služi za prikaz stanja OXOCell objekta unutar ćelije, što se realizuje kroz metodu fillCell (Listing 13). Ova metoda omogućava prikaz stanja ćelije uz odgovarajuću vizuelnu reprezentaciju. Takođe ovde omogućavamo obradu događaja klikom na ćeliju preko metode addEventHandlerOn prosleđujući klasu BlClickEvent (Ducasse, Demeulenaere i Dias 2024.).

```
OXOCellElement >> 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

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

Listing 13: Metoda fillCell

Metoda click (Listing 14), 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 sekciji za pokretanje igre. Ukoliko se klikne već popunjena ćelija, generiše se upozorenje, a ako nije, prave se potezi igrača i računara preko metoda makeMove (Listing 15) i makeAIMove (Listing 16).

```
OXOCellElement >> 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

```
OXOCellElement >> 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.

```
OXOCellElement >> 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 (Listing 17). Ako se ispostavi da je igra završena, sve ćelije se boje odgovarajućom bojom koristeći metodu changeChildrenColor koja se može naći u OXOGridElement klasi.

```
OXOCellElement >> 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 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. (Listing 18)

```
OXOGridElement >> 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. Ona služi za postavljanje OXOCellElement objekata za svaki OXOCell iz OXOGrid objekta koji predstavlja naš model. (Listing 19)

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

Listing 19: OXOGridElement setter metoda za model

Takođe ovde imamo changeChildrenColor (Listing 20) metodu koju smo već pomenuli i koja menja početnu konfiguraciju boje za svaki objekat klase OXOCellElement.

```
OXOGridElement >> changeChildrenColor: aColor
    self childrenDo: [ :x | x background: aColor ]
```

Listing 20: Metoda changeChildrenColor

Da bi napravili razliku između inicijalizacije OXOGridElement objekta sa uključenom i isključenom veštačkom inteligencijom, konstruišemo dve metode na nivou klase. (Listing 21 i 22)

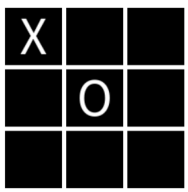
```
OXOGridElement class >> new: aBlSpace
    ↑ self new game: aBlSpace
```

Listing 21: Klasna metoda za kreiranje OXOGridElement objekta

```
OXOGridElement class >> newWithAI: aBlSpace
    ↑ (self new game: aBlSpace) ai: true
```

Listing 22: Klasna metoda za kreiranje uz uključenu veštačku inteligenciju

Na slici 1 se može videti kako izgleda rezultat kôda za GUI.



Slika 1: Pregled GUI-a

5 Pokretanje igre

OXO-Game paket sadrži OXO klasu sa varijablama instance, game i space. Ona sopstvenim instanciranjem instancira BlSpace objekat iz Bloc razvojnog okvira pod sopstvenom space varijablom. BlSpace objekat predstavlja novi sistemski prozor na koji možemo da dodajemo BlElement objekte (Ducasse, Demeulenaere i Dias 2024.). Instanciramo OXO preko njenih metoda na nivou klase playVsPlayer ili playVsComputer koje dodaju BlSpace elementu adekvatan OXOGridElement objekat inicijalizovan preko jednostavne new ili newWithAi metode koje se realizuju u OXOGridElement klasi kao klasne metode. Na listingu 23 možemo videti metodu playVsPlayer metodu koja će instancirati naš progam.

```
OXO class >> 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 na listingu 24. Ne menjamo space varijablu, već samo inicijalizujemo novi game na osnovu toga koji je tip igre pokrenut.

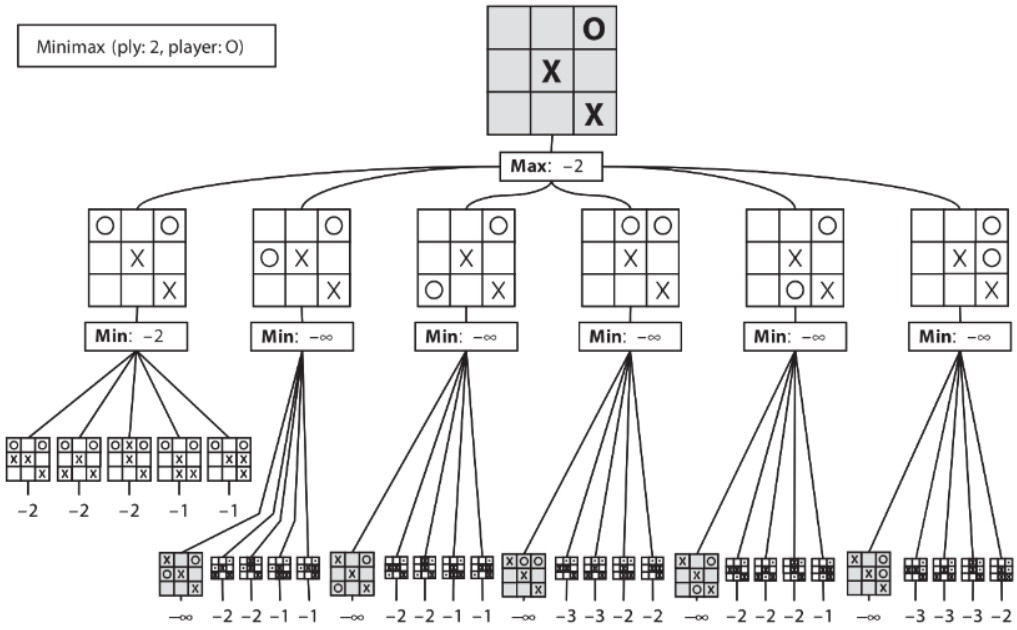
```
OXO >> reset
    game := (self space root childAt: 1) ai
        ifTrue: [ OXOGridElement newWithAI: self ]
        ifFalse: [ OXOGridElement new: self ].
    space root removeChildren.
    space root addChild: game
```

Listing 24: Metoda za resetovanje partije igre

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

6 Izrada minimaks algoritma

6.1 Objašnjenje algoritma



Slika 2: Primer minimaks pretrage (Heineman, Pollice i Selkow 2008.)

Minimaks algoritam jeste rekurzivan algoritam za odabir sledećeg najboljeg poteza u igrama sa n brojem igrača (Heineman, Pollice i Selkow 2008.). Oslanjamo se na generisanje stabla koje pokriva sve moguće scenarije u igri. Algoritam će biti opisan u kontekstu „iks-oks“ igre kao što se može videti na slici 2. 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 i Norvig 2009.). 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. Postoje raznorazne varijacije ovog algoritma kroz koje nećemo prolaziti u ovom radu. Pretragu takođe možemo skratiti proveravanjem koliko smo duboko izvršili pretragu u stablu. U slučaju standardne „iks-oks“ igre je to jednostavno iz razloga što broj scenarija ne prelazi vrednost od 255 168 (Bottomley 2001.) kako nam snaga modernog računara dopušta taj nivo procesiranja. Neke varijacije nude optimizacije kao što su alfa-beta pretraga koja bi dosta ubrzala proces pretrage u kompleksnijim igrama gde bi se odmah odbacile pretrage grana za koje znamo sigurno da nisu povoljne. (Heineman, Pollice i Selkow 2008.; Russell i Norvig 2009.)

Efikasnost algoritma zavisi dosta od načina na koji proveravamo povoljnost scenarija po igrače. Određivanje povoljnosti je veoma kompleksan proces u slučaju da su igra i pravila iste kompleksna. Za rešenje ovog problema se moramo oslanjati na heurističke metode (Heineman, Pollice i Selkow 2008.; Russell i Norvig 2009.). Za standardnu „iks-oks“ igru je lako odrediti povoljnost scenarija kako je igra jednostavna i ne moramo skraćivati pretragu.

6.2 Implementacija algoritma

Pre implementacije minimaksa se mora prvo izvršiti implementacija metoda koje će nam resetovati stanja prilikom rekurzije koju budemo izvršavali kako ne bi poremetili originalni model koji prosleđujemo. Potrebne su nam metode undoMove koja će pripadati klasi OXOGrid (Listing 26) i reset koja će pripadati klasi OXOCell (Listing 27).

```
1 OXOGrid >> undoMove
2     moves := moves - 1
```

Listing 25: Pomoćna metoda undoMove

```
1 OXOGrid >> undoMove: aCell
2     aCell reset.
3     self undoMove.
```

Listing 26: Metoda undoMove

```
1 OXOCell >> reset
2     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 board koja predstavlja trenutni scenario igre. Pri instanciranju će se proslediti trenutno popunjen model igre u tu varijablu. Implementiramo metodu makeBestMove (Listing 28) koja će pokrenutni minimax metodu za svaku ćeliju u našem modelu i na osnovu toga će se izvući ćelija sa najvećom težinskom vrednošću.

```
OXOAI >> 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 (Listing 29) 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 (Listing 30) i max (Listing 31) se upravo to radi i sa njima rekurzivno zalazimo dublje kroz svaki mogući scenario dok nam minimaks algoritam ne vrati vrednost koja će označavati pobedu, gubitak ili nerešeno stanje.

```
OXOAI >> 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

```
OXOAI >> 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

Minimizujuća metoda je slična kao maksimizujuća gde se vrši „obrnut“ proces rada.

```
OXOAI >> 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 (Listing 32) vrši najbitniju funkciju jer ona opisuje heurističnu metodu koju koristimo za određivanje povoljnosti scenarija do kog je pretraga došla. Ona se oslanja na već odrađene metode u modelu koje vrše prostu proveru da li ima pobednika ili ne. Da rezimiramo, 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.

```
OXOAI >> 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. Trenutna implementacija nema restrikciju u pretrazi tako da će rezultat igara u najboljem slučaju po ljudskog igrača biti nerešen.

7 Budućnost rada

Rad se može dalje unaprediti skraćivanjem i sklanjenjem redudantnog kôda koji je posledica raznog testiranja. Nakon toga se čitav projekat može pripremiti za distribuciju preko Pharo podržanog menadžera za pakete. Model igre se može sa lakoćom zameniti modelima drugih igara sa nultom sumom i adekvatno skalirati što je bilo jedan od glavnih ciljeva ovog rada.

8 Zaključak

U ovom radu smo uspešno realizovali klasičnu verziju igre „iks-oks“. 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.

Literatura

Bottomley, Henry (2001.). *How many Tic-Tac-Toe (noughts and crosses) games? — se16.info*. `http://www.se16.info/hgb/tictactoe.htm`. Pristupljeno: 2020–05-24.

Ducasse, Stéphane (2000.). *Object-Oriented Design with Smalltalk — a Pure Object Language and its Environment*. `https://scg.unibe.ch/archive/lectures/DucasseLectures/Duca00y1SmalltalkLectures.pdf`. Pristupljeno: 2020–05-30.

Ducasse, Stéphane, E. Demeulenaere i M. Dias (2024.). *A memory game: A simple tutorial with Bloc*. `https://books.pharo.org/booklet-ASimpleMemoryGameInBloc/2024-06-05-ASimpleBlocTutorial.pdf`. 12/14 rond-point des Champs-Élysées,75008 Paris.

Ducasse, Stéphane, Gordana Rakić i Sebastijan Kaplar (mar. 2022.). *Pharo 9 by example*. Books on Demand.

Gamma, Erich i dr. (okt. 1994.). *Design patterns*. Boston, MA: Addison Wesley.

Heineman, George T, Gary Pollice i Stanley Selkow (okt. 2008.). *Algorithms in a Nutshell*. Sebastopol, CA: O'Reilly Media.

Russell, Stuart i Peter Norvig (dec. 2009.). *Artificial intelligence*. 3. izdanje. Upper Saddle River, NJ: Pearson.

Wang, Hao (dec. 1993.). *Popular lectures on mathematical logic*. Dover Books on Mathematics. Mineola, NY: Dover Publications.