Inteligentni agent za igranje šaha

Dokumentacija

Autori:

Mihailo Trajković RN 3/20
Ivan Jevtić RN 4/20

Struktura rešenja

Inteligentni agent implementiran je u programskom jeziku Java, interfejs pomoću kojeg korisnik učestvuje u igri realizovan je uz pomoć biblioteke Swing.

Sama mehanika igre je implementirana nezavisno od ostatka projekta, u okviru paketa *engine*. Tu se izdvajaju 3 klase: *Board*, *Move* i *Piece*.

Za tok konkretne partije između dva igrača, u našem slučaju između igrača i mašine zadužena je klasa *Game*

Klasa Board

Polja:

```
private Piece[][] squares;
private List<Piece> pieces;
private Stack<Move> moves;
private int boardValue = 0;
private Piece whiteKing;
private Piece blackKing;
```

Matrica *squares* sadrži informacije o rasporedu figura na tabli. Matrica je dimentija 8x8 i inicijalizuje se u konstruktoru. Ukoliko se na nekom polju ne nalazi figura, vrednost odgovarajućeg polja u matrici je *null*.

Celobrojna promenljiva boardValue predstavlja sumu vrednosti figura koje se trenutno nalaze na tabli, vrednosti figura su 100, 300, 300. 500, 1000 za pešaka, skakača, lovca, topa i damu redom. Vrednosti crnih figura imaju predznak minus, pa je vrednost početne pozicije 0. Ovaj podatak će biti potreban agentu kako bi mogao da proceni koje pozicije su povoljne za njega, a koje ne.

whiteKing i *blackKing* su polja koja služe efikasnijoj proveri da li je kralj pod šahom.

Metode:

Nazivi samih metoda poprilično dobro označavaju šta je njihova svrha.
Metoda *makeMove* menja stanje table igranjem zadatog poteza, dok metoda *takeBackMove* vraća tablu u stanje pre nego što je određeni potez odigran.
Metoda *getAllMoves* vraća listu mogućih poteza u trenutnoj poziciji.

```java
public void makeMove(Move move) {...}

public void takeBackMove(Move move){...}

public Piece getPiece(int x, int y) { return squares[x][y]; }

public int getRating() { return boardValue; }

public int getAwardPosition(Piece p) {...}

private int getPieceValue(Piece p) {...}

public Deque<Move> getAllMoves(boolean color) {...}

public boolean insideTable(int x, int y) { return x >= 0 && y >= 0 && x < 8 && y < 8; }

public boolean can(Piece p, int x, int y) {...}

public boolean freeSquare(int x, int y) { return insideTable(x,y) && (getPiece(x,y) == null); }

public boolean enemyPiece(Piece p, int x, int y){...}

public boolean enemyPiece(boolean c, int x, int y){...}

public boolean isCheck(boolean color) {...}

public boolean isAttackedSquare(int x, int y, boolean color){...}
```

Klasa Move

Ova klasa sadrži neophodne informacije da bi se jedan potez odigrao, odnosno vratio po potrebi. Klasa implemenita interfejs Comparable, i osim svoje implementacije metode *compareTo*, *toString* i getera i setera nema drugih metoda.

```java
private Piece piece;
private int endX;
private int endY;
private int startX;
private int startY;

private Piece capturedPiece;
private boolean oldMoved = false;

private boolean captures;

private Move castleMove1;
private Move castleMove2;
```

Klasa Piece

Ova klasa je apstraktna i njene konkretne implementacije su klase Pawn, Knight, Bishop, Rook, Queen i King.

Polja:

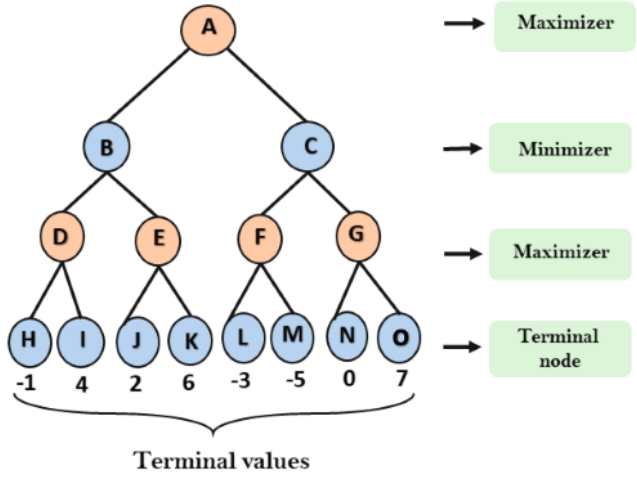
```
Board board;
String name;
boolean color;
int x;
int y;
int value;
boolean moved = false;
```

Klasa Game

```
private Player player1;
private Player player2;
private Board board;
private Player playerOnMove;
private BoardView boardView;

public Game() { this.board = new Board(); }

public void setPlayers(Player player1, Player player2){...}

public void startGame(){...}

public void makeMove(Player player, Move move) {...}

public void changeTurn() {...}

private void setOnMove(Player player){...}
```

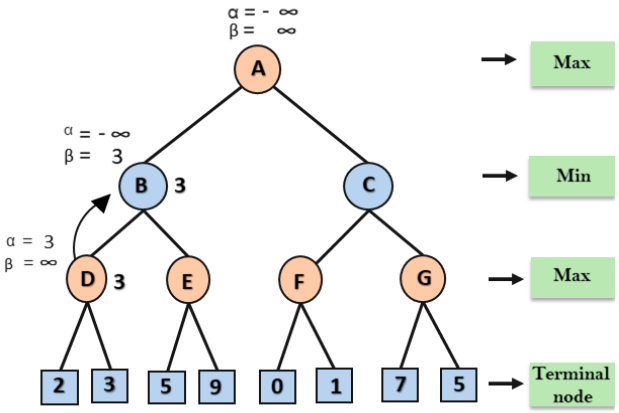
Klasa Player je apstraktna i nju realzuju klase *Human* i *Bot* implementacijom metode *playMove*. Detaljnije o implemetaciji ove pmetode u klasi Bot, odnosno algoritmu po kome radi naš inteligentni agent u nastavku.

Igranje poteza inteligentnog agenta igrača



Inteligentni agent i njegovo igranje se zasniva na algoritmu Minimaks. Cilj je da kompjuter iskoristi svoje resurse, tako što će simulirati sve moguće poteze, do neke određene dubine, i na taj način videti koji mu se potez najviše isplati.
Minimaks algoritam podrazumeva da agent proceni korisnost akcije sa pretpostavkom da protivnički agent igra optimalno. On pokušava da u najgorem slučaju izvuče najveću korisnost. Algoritam se zasniva na međusobnoj promeni aktera. Agent donosi odluku na osnovu toga šta bi njegov protivnik mogao da odgovori, pa šta bi on odgovorio na taj odgovor itd.

Problem kod ovog pristupa je složenost, jer običan minimaks algoritam radi u eksponencijalnoj složenosti, pa dubina do koje agent može da ide je veoma mala(par poteza unapred). Način na koji možemo smanjiti složenost minimaksa, a tako doći i do veće dubine nas dovodi do algoritma Minimaks sa Alfa-Beta odsecanjem. Razlika između Minimaksa i ovog algoritma, je činjenica da ne moramo da proveravamo sve moguće grane, već možemo odseći neke grane, koje neće uticati na krajnji rezultat. Na primer, ako je Min igrač našao već rezultat 5, a u trenutoj grani igrača Max nalazimo rezultat 7, znamo da će Max igrač sigurno uzeti vrednost barem 7, i na osnovu toga Min ga sigurno neće birati, tako da možemo završiti pretragu u trenutnoj grani, i vratiti rezultat 7 kao odgovor igrača Max.

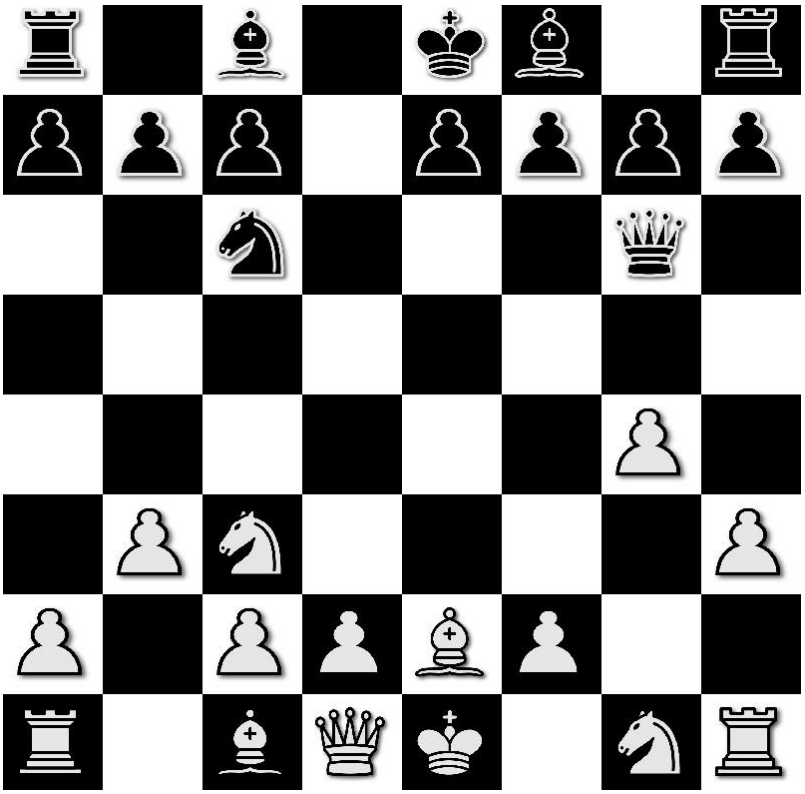


```
private AlfaBeta minimax(Board b, int alfa, int beta, boolean color, int depth, boolean jeo) {
    int res = 10000;
    if(!color) res = -10000;
    Deque<Move> moves = b.getAllMoves(color);
    if(depth == 0) {
        if(jeo) return minimaxJede(b,alfa,beta,color, duz: 0, jeo: true);
        return new AlfaBeta(b.getRating(), alfa, beta);
    }
    for (Move move : moves) {
        b.makeMove(move);
        int p = minimax(b, alfa, beta, !color, depth: depth - 1, move.isCaptures()).p;
        b.takeBackMove(move);
        if (!color) {
            if (p > res) res = p;
            if (res >= beta) return new AlfaBeta(res, alfa, beta); //odsecanje
            alfa = Math.max(alfa, res);
        } else {
            if (p < res) res = p;
            if (res <= alfa) return new AlfaBeta(res,alfa,beta); //odsecanje
            beta = Math.min(beta, res);
        }
    }
    return new AlfaBeta(res,alfa,beta);
}
```

Sada možemo još malo da optimizujemo ovaj algoritam. Isplati se igrati prvo bolje poteze, da bismo kasnije imali više odsecanja. Tako da će agent prvo igrati poteze koji mu najviše poboljšavaju poziciju.

```
@Override
public int compareTo(Move o) {
    if(this.captures && o.captures) {
        if(this.piece.getValue() - this.capturedPiece.getValue() > o.piece.getValue() - o.capturedPiece.getValue())
            return 1;
        return -1;
    }
    if(this.captures) return 1;
    return -1;
}
```

Drugi problem kod ovog algoritma je taj da agent nema uvid u igru van zadate dubine, pa neki potez koji naizgled može delovati dobro kada se gleda x poteza unazad, može biti loš na nekoj dubini k, k>x.



Imamo situaciju na slici, nalazimo se na maksimalnoj mogućoj dubini, crni je na potezu(inteligentni agent). Agent može da odluči da kraljicom pojede nekog od pešaka na C2 ili G4. Pošto se pretraga završava na toj dubini, taj potez deluje dobro. Ali već u sledećem potezu, beli uzima crnu kraljicu, i ispostavlja se da je pređašnji potez zapravo bio loš. Tako da se nećemo zaustavljati kada dođemo do krajnje dubine, već ćemo pokrenuti novi minimaks algoritam, koji razmatra samo poteze koji dovode do jedenja. Tako ćemo poboljšati agentovo procenjivanje situacije. Taj novi algoritam se zaustavlja kada probani potez nije doveo do uzimanja protivničke figure, i gleda najbolju moguću situaciju iz nje. Često se dešava da taj novi minimaks algoritam ima mnogo mogućnosti u sebi, od kojih velika većina nije verovatna, tako da ćemo dodati ograničenja kao što su: gledanje do dubine 5(nije verovatno da će se pojesti uzastopno više od 5 figura), razlika preko 1100 u pozicijama(ovo je jako velika prednost i malo je verovatna, a ako je već ovolika prednost između igrača, pobednik je praktično odlučen).

```java
private AlfaBeta minimaxJede(Board b, int alfa, int beta, boolean color, int duz, boolean jeo) {
    int res = b.getRating();
    if (!color) {
        if (res >= beta) return new AlfaBeta(res, alfa, beta); //odsecanje
        alfa = Math.max(alfa, res);
    } else {
        if (res <= alfa) return new AlfaBeta(res,alfa,beta); //odsecanje
        beta = Math.min(beta, res);
    }
    Deque<Move> moves = b.getAllMoves(color);
    if(moves.isEmpty() || !moves.getFirst().isCaptures() || duz > 5 || Math.abs(b.getRating()) > 1100) {
        return new AlfaBeta(b.getRating(), alfa, beta);
    }
    int i = 0;
    List<Move> niz = new ArrayList<>(moves);
    Collections.sort(niz);
    for(Move move: niz) {
        b.makeMove(move);
        int p;
        if(jeo) p = minimaxJede(b, alfa, beta, !color, duz: duz+1, move.isCaptures()).p;
        else p = b.getRating();
        b.takeBackMove(move);
        if (!color) {
            if (p > res) res = p;
            if (res >= beta) return new AlfaBeta(res, alfa, beta); //odsecanje
            alfa = Math.max(alfa, res);
        } else {
            if (p < res) res = p;
            if (res <= alfa) return new AlfaBeta(res,alfa,beta); //odsecanje
            beta = Math.min(beta, res);
        }
    }
```