SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Dominik Maloić

VIŠEAGENTNI SUSTAV ZA SIMULACIJU IGRE ŠAHA

PROJEKT

VIŠEAGENTNI SUSTAVI

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dominik Maloić

Matični broj: 0016131955

Studij: Informacijsko i programsko inženjerstvo

VIŠEAGENTNI SUSTAV ZA SIMULACIJU IGRE ŠAHA

PROJEKT

Mentor:

dr. sc. Bogdan Okreša Đurić

Varaždin, siječanj 2024.

Dominik Maloić

Izjava o izvornosti

Izjavljujem da je ovaj projekt izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

Projekt pod nazivom "Višeagentni sustav za simulaciju igre šaha" istražuje implementaciju kompleksnog višeagentnog sustava koji simulira igru šaha. Sustav koristi umjetnu inteligenciju, točnije Minimax algoritam, za donošenje odluka o potezima unutar šahovske igre. Cilj projekta je razviti inteligentnog agenta koji može djelovati kao protivnik šahovskog igrača, pružajući izazovne i raznolike poteze. Teorijsko-metodološka polazišta istražuju temeljne koncepte umjetne inteligencije, posebno Minimax algoritma, koji omogućuje agentima donošenje optimalnih poteza u igrama s nul-sumnim karakteristikama, kao što je šah. Osim toga, projekt se fokusira na implementaciju višeagentnog sustava, gdje se svaki agent ponaša kao inteligentni entitet sposoban procijeniti trenutno stanje igre i donositi odluke u skladu s definiranim pravilima. Glavne teze rada obuhvaćaju analizu performansi Minimax algoritma u kontekstu igre šaha te pružaju uvid u funkcionalnosti višeagentnog sustava. Implementacija ovog sustava pruža korisnicima mogućnost igranja protiv inteligentnog virtualnog protivnika, omogućujući im unapređenje vještina u šahu. Zaključci projekta naglašavaju važnost umjetne inteligencije u kontekstu strateških igara poput šaha te ističu mogućnosti daljnjeg poboljšanja i proširivanja sustava. Projekt predstavlja korak prema razvoju sofisticiranih višeagentnih sustava u simulaciji šahovske igre, otvarajući vrata daljnjim istraživanjima u području umjetne inteligencije i igara.

Ključne riječi: Višeagentni sustav;Simulacija igre šaha;Umjetna inteligencija;Minimax algoritam;Strateška igra;Virtualni protivnik;Razvoj igara

Sadržaj

1. Uvod

U ovom projektu istražuje se implementacija višeagentnog sustava s naglaskom na simulaciju igre šaha. Projekt koristi umjetnu inteligenciju, posebno Minimax algoritam, za donošenje odluka o potezima unutar šahovske igre. Glavni cilj projekta jest razviti inteligentnog virtualnog protivnika koji pruža izazovne i raznolike poteze kako bi poboljšao vještine šahovskih igrača.

Šah je jedna od najpoznatijih i najigranijih igra na ploči, rijetko koja osoba nije iskušala svoje znanje i vještine u igri šaha, a oni koji jesu znaju kako izazovna ta igra može biti. Od raznih otvaranja do mogučih žrtvovanja figura, postoji mnogo različitih načina igranja šaha. Ja sam odlučio napraviti agenta koji će igrati šah prema minimax algoritmu.

2. Metode i tehnike rada

Projekt je u potpunosti izrađen u programskom jeziku python, od pomoćnih biblioteka korišten je python-chess [1] biblioteka za rad sa šahovskim pravilima i potezima. Razvojno okruženje koje sam koristio bilo je MintVAS slika za VirtualBox, program je pisan u VSCode.

Od metoda istraživanja koristit ćemo kombinaciju deskriptivne analize i eksperimentalnog pristupa kako bismo proučili teorijske osnove umjetne inteligencije, posebice Minimax algoritma, te primijenili dobivene spoznaje na konkretnu simulaciju igre šaha.

Provodit ćemo dubinsko istraživanje Minimax algoritma, analizirajući njegove prednosti, nedostatke i primjene u igrama s nul-sumnim karakteristikama. Razviti ćemo simulacijski model igre šaha koji će koristiti višeagente, pridržavajući se pravila igre. Provest ćemo eksperimente kako bismo testirali performanse razvijenog sustava, prikupljajući podatke o brzini odlučivanja, kvaliteti poteza te reakcijama na različite igračke strategije. U ovom poglavlju treba opisati koje će metode i tehnike biti korištene pri razradi teme, kako su provedene istraživačke aktivnosti, koji su programski alati ili aplikacije korišteni.

Validacija sustava provodit će se kroz usporedbu rezultata simulacije s očekivanim rezultatima prema pravilima igre šaha. Analiza rezultata uključivat će statističku obradu podataka o performansama Minimax algoritma i višeagentnog sustava.

3. Razrada teme

3.1. Minmax algoritam u kontekstu šahovske igre

Minimax algoritam je klasičan pristup odlučivanju u igrama s nul-sumnim karakteristikama, a njegova primjena u igri šaha donosi duboko razumijevanje i učinkovite strategije. Ovo poglavlje detaljno će razraditi Minimax algoritam, njegove osnovne koncepte te način primjene u simulaciji šahovske igre.

3.1.1. Osnovni Koncepti Minimax Algoritma

Minimax algoritam je algoritam pretraživanja prostora stanja u igrama, a njegova osnovna svrha je maksimizirati korist jednog igrača, dok istovremeno minimizira korist protivničkog igrača. Ključna pretpostavka je da oba igrača djeluju racionalno, te da svaki igrač donosi odluke koje će mu najviše povećati korist u igri. [2]

Minimax algoritam se često predstavlja stablom odlučivanja, gdje čvorovi predstavljaju stanja igre, a grane predstavljaju moguće poteze.

Algoritam ima dva tipa čvorova - "maximizer" čvorove koji predstavljaju poteze igrača koji trenutno pokušava maksimizirati svoju korist, i "minimizer" čvorove koji predstavljaju poteze protivničkog igrača.

3.1.2. Primjena Minimax Algoritma u Šahovskoj Igri

U kontekstu igre šaha, Minimax algoritam može biti implementiran na sljedeći način, preko stanja igre, heuristike i pomoću rekurzivnog pretraživanja stabla odlučivanja. Svako stanje igre predstavlja trenutno raspored figura na šahovnici i informacije o trenutnom igraču na potezu.

Razvijanje funkcije vrednovanja ili heuristike koja ocjenjuje vrijednost pozicije na šahovnici. Ova funkcija pomaže algoritmu procijeniti prednost ili nedostatak određenog rasporeda figura.

Algoritam rekurzivno pretražuje stablo odlučivanja, simulirajući poteze oba igrača sve do zadanog dubinskog nivoa. Na temelju rezultata, donosi se optimalan potez.

3.1.3. Izazovi i Optimizacije

Odabir odgovarajuće dubine pretraživanja ključan je faktor. Veća dubina pruža preciznije, ali i zahtjevnije izračune. Za najbolji omjer odluka i brzine odlučio sam koristiti dubinu 4.

Korištenje algoritama obrezivanja poput Alpha-Beta pruningu za smanjenje broja evaluacija i ubrzanje procesa.

3.2. Izrada simulirane igre šaha

U ovom poglavlju objasnit ću programski kod i implementirane funkcije sustava za simuliranje igre šaha, strukturu programa te optimizacije koje se mogu primijeniti.

3.2.1. Funkcija za evaluiranje ploče

```
def evaluate_board(board):
    evaluation = 0
    for square in chess.SQUARES:
        piece = board.piece_at(square)
        if piece is None:
            continue
        value = {
            chess.PAWN: 10,
            chess.KNIGHT: 30,
            chess.BISHOP: 30,
            chess.ROOK: 50,
            chess.QUEEN: 90,
            chess.KING: 900
        }[piece.piece_type]

        value = value if piece.color == chess.WHITE else -value
        evaluation += value

    evaluation += 0.1 * len(list(board.legal_moves))

    return evaluation
```

Program koristi funkciju vrednovanja pozicije `evaluate_board` koja dodjeljuje ocjene različitim figurama na šahovnici te potiče poteze koji povećavaju dostupnost poteza.

Inicijalizacija ocjene:

```
evaluation = 0
```

Inicijalizacija varijable `evaluation` na nulu. Ova varijabla će se koristiti za akumulaciju numeričke vrijednosti šahovske pozicije.

Petlja po kvadratima:

```
for square in chess.SQUARES:
```

Ova petlja iterira kroz sve kvadrate na šahovnici. `chess.SQUARES` je lista svih mogućih kvadrata na standardnoj šahovnici.

Dohvati Figuru na Kvadratu:

```
piece = board.piece_at(square)
```

Za svaki kvadrat, dohvaća se šahovska figura na tom kvadratu koristeći `piece_at` metodu objekta `board`.

Preskoči prazne kvadrate:

```
if piece is None:
    continue
```

Ako je kvadrat prazan (nema figure), preskače se ostatak petlje i prelazi na sljedeću iteraciju.

Dodijeli vrijednosti figurama:

```
value = {
    chess.PAWN: 10,
    chess.KNIGHT: 30,
    chess.BISHOP: 30,
    chess.ROOK: 50,
    chess.QUEEN: 90,
    chess.KING: 900
}[piece.piece_type]
```

Dodjeljuje numeričku vrijednost varijabli `value` na temelju tipa šahovske figure na kvadratu. Vrijednosti su tipične heurističke vrijednosti koje predstavljaju relativnu snagu svake figure.

Prilagodi vrijednost na temelju boje figure:

```
value = value if piece.color == chess.WHITE else -value
```

Ako je figura bijela, vrijednost ostaje nepromijenjena. Ako je figura crna, vrijednost se negira. Ova prilagodba uzima u obzir da figure imaju pozitivne vrijednosti za bijele i negativne za crne.

Akumuliraj vrijednosti:

```
evaluation += value
```

Prilagođena vrijednost dodaje se varijabli `evaluation`.

Poticanje algoritma na određene poteze:

```
evaluation += 0.1 * len(list(board.legal_moves))
```

Dodaje se dodatni izraz u ocjenu na temelju broja dostupnih legalnih poteza. Ovo potiče algoritam da preferira pozicije s više dostupnih poteza.

Vrati konačnu ocjenu:

```
return evaluation
```

Funkcija vraća konačni rezultat ocjene za šahovsku poziciju.

Ukratko, ova funkcija dodjeljuje numeričke vrijednosti figurama na ploči, prilagođava za boju figure i razmatra dostupnost legalnih poteza kako bi ocijenila ukupnu snagu određene šahovske pozicije. Konkretni brojevi dodijeljeni svakoj figuri temelje se na heurističkim razmatranjima, a cilj funkcije je pružiti kvantitativnu mjeru željenosti pozicije.

3.2.2. Implementacija Minimax Algoritma u Programu

```
def minimax(board, depth, maximizing_player):
    if depth == 0 or board.is_game_over():
        return evaluate_board(board)

    legal_moves = list(board.legal_moves)

    if maximizing_player:
        max_eval = float('-inf')
        for move in legal_moves:
            board.push(move)
            eval = minimax(board, depth - 1, False)
            board.pop()
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for move in legal_moves:
            board.push(move)
            eval = minimax(board, depth - 1, True)
            board.pop()
            min_eval = min(min_eval, eval)
        return min_eval
```

Funkcija `minimax` predstavlja srce Minimax algoritma. Rekurzivno pretražuje stablo odlučivanja, simulirajući poteze oba igrača i računajući vrijednost svake pozicije.

Funkcija `find_best_move` odabire najbolji potez pomoću Minimax algoritma, uzimajući u obzir dubinu pretraživanja. Prikazuje razmatrane poteze i njihove evaluacije.

```
def minimax(board, depth, maximizing_player):
```

Definicija funkcije minimax koja prima tri argumenta: `board` predstavlja trenutno stanje igre, `depth` predstavlja dubinu pretrage (koliko poteza unaprijed gledamo), a `maximizing_player` označava je li trenutni igrač maksimizirajući ili minimizirajući.

```
if depth == 0 or board.is_game_over():
    return evaluate_board(board)
```

Provjera jesmo li dosegli maksimalnu dubinu pretrage ili je igra završena. Ako jest, vraća se rezultat evaluacije trenutnog stanja ploče pomoću funkcije `evaluate_board`.

```
legal_moves = list(board.legal_moves)
```

Dohvačanje svih legalnih poteza za trenutno stanje ploče.

```
if maximizing_player:
    max_eval = float('-inf')
    for move in legal_moves:
        board.push(move)
        eval = minimax(board, depth - 1, False)
        board.pop()
        max_eval = max(max_eval, eval)
    return max_eval
```

Ako je trenutni igrač maksimizirajući, funkcija iterira kroz sve legalne poteze, simulira ih na ploči (`board.push(move)`), poziva minimax rekuzivno za smanjenje dubine (`minimax(board, depth - 1, False)`), vraća se na prethodno stanje ploče (`board.pop()`), te ažurira maksimalnu vrijednost (`max_eval`) prema dobivenoj evaluaciji.

```
else:
    min_eval = float('inf')
    for move in legal_moves:
        board.push(move)
        eval = minimax(board, depth - 1, True)
        board.pop()
        min_eval = min(min_eval, eval)
    return min_eval
```

Ako je trenutni igrač minimizirajući, postupak je sličan, ali se ažurira minimalna vrijednost (`min_eval`).

3.2.3. Funkcija pronalaženja najboljeg koraka

Ovaj kod koristi minimax algoritam za evaluaciju svakog poteza na temelju dubine pretrage i odabire najbolji potez na temelju najviše evaluacije. Ispisi su dodani radi praćenja ponašanja algoritma.

```
def find_best_move(board, depth):
    legal_moves = list(board.legal_moves)
    best_move = None
    best_eval = float('-inf')

    for move in legal_moves:
        board.push(move)
        eval = minimax(board, depth - 1, False)
        board.pop()

        print(f"Potez: {move.uci()}, Evaluacija: {eval}")

        if eval > best_eval:
            best_eval = eval
            best_move = move

        print(f"Najbolji potez: {best_move.uci()}, Najbolja evaluacija:
        ↪  {best_eval}")

    return best_move
```

```
def find_best_move(board, depth):
```

Definiranje funkcije `find_best_move` koja traži najbolji potez na temelju minimax algoritma. Funkcija prima `board` - trenutno stanje igre, i `depth` koji predstavlja dubinu pretrage.

```
best_move = None
best_eval = float('-inf')
```

Inicijalizira varijable `best_move` i `best_eval`. `best_move` će sadržavati najbolji potez, a `best_eval` najbolju evaluaciju. Početna vrijednost `best_eval` je postavljena na negativnu beskonačnost.

```
for move in legal_moves:
    board.push(move)
    eval = minimax(board, depth - 1, False)
    board.pop()

    print(f"Potez: {move.uci()}, Evaluacija: {eval}")
```

```
    if eval > best_eval:
        best_eval = eval
        best_move = move

    print(f"Najbolji potez: {best_move.uci()}, Najbolja evaluacija: {best_eval}")
```

Započinje petlja koja prolazi kroz sve legalne poteze. Unutar petlje, svaki potez se privremeno simulira na ploči. Poziva se minimax funkcija smanjujući dubinu pretrage, a rezultat se sprema u varijablu `eval`. Nakon simulacije, vraćamo ploču na prethodno stanje s `board.pop()`. Ispisuje informacije o trenutnom potezu i njegovoj evaluaciji.

Ako trenutna evaluacija (`eval`) nadmašuje trenutnu najbolju evaluaciju (`best_eval`), ažuriramo `best_eval` i `best_move` s novim najboljim potezom.

Ispisuje informacije o trenutno najboljem potezu i njegovoj evaluaciji. Nakon završetka petlje, vraća potez koji se smatra najboljim prema minimax algoritmu.

3.2.4. Funkcija provjere legalnosti pokreta

Iako agent uvijek zna legalnost pokreta potrebna mi je bila jedna funkcija koja provjerava pokrete igrača.

```
def is_legal_move(board, move):
    legal_moves = list(board.legal_moves)
    return move in legal_moves
```

3.2.5. Main metoda

Funkcija Main u kojoj je moguće mijenjanje dubine minimax algoritma te postavlja ploču i obrađuje unose korisnika i poziva funkcije agenta.

```
def main():
    board = chess.Board()
    depth = 4  # Moguća je promjena dubine, veći broj označava dulje vrijeme pokreta

    while not board.is_game_over():
        print(board)
        if board.turn == chess.WHITE:
            legal_moves = list(board.legal_moves)
            moves_list = " ".join(move.uci() for move in legal_moves)
            print("Legalni potezi:", moves_list)
            user_move = input("Upišite svoj potez (iz liste mogucih): ")
            if is_legal_move(board, chess.Move.from_uci(user_move)):
                board.push(chess.Move.from_uci(user_move))
            else:
```

```
                print("Nedozvoljen potez. Pokušajte ponovo.")
        else:
            print("AI razmišlja...")
            best_move = find_best_move(board, depth)
            print("AI potez:", best_move)
            board.push(best_move)

    print("Game Over!")
    print("Rezultat: {}".format(board.result()))
```

Petlja se izvršava sve dok igra nije gotova, nakon svakog koraka ispisuje se trenutno stanje šahovske ploče. Provjerava čiji je red za potez. Ako je bijeli igrač na potezu, omogućuje mu unos poteza.

```
legal_moves = list(board.legal_moves)
moves_list = " ".join(move.uci() for move in legal_moves)
print("Legalni potezi:", moves_list)
```

Ispisuje listu svih legalnih poteza koje bijeli igrač može odabrati.

```
user_move = input("Upišite svoj potez (iz liste mogućih): ")
if is_legal_move(board, chess.Move.from_uci(user_move)):
    board.push(chess.Move.from_uci(user_move))
else:
    print("Nedozvoljen potez. Pokušajte ponovo.")
```

Omogućuje bijelom igraču unos vlastitog poteza i provjerava je li taj potez legalan. Ako jest, potez se izvodi, inače se ispisuje poruka o nedozvoljenom potezu.

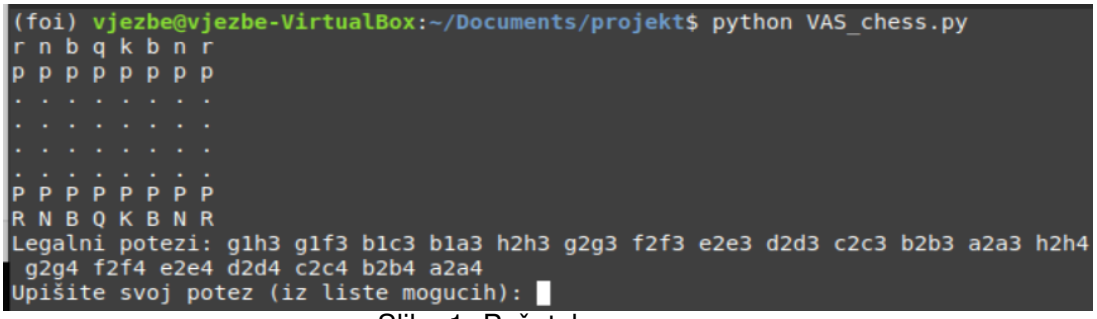
```
else:
            print("AI razmišlja...")
            best_move = find_best_move(board, depth)
            print("AI potez:", best_move)
            board.push(best_move)
```

Ako crni igrač (AI) ima potez, AI razmišlja o najboljem potezu koristeći minimax algoritam.Prikazuje poruku o razmišljanju AI i izvodi najbolji potez koji je pronađen pomoću funkcije `find_best_move`. Na kraju igre ispisuje se poruka o završetku igre i prikazuje se rezultat.

3.3. Rad programa

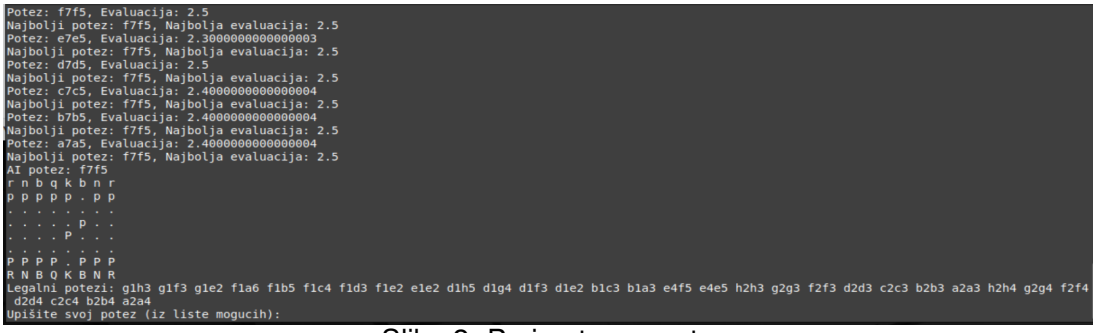
U ovom poglavlju prikazati ću rad programa sa slikama rada.

Program pokrećemo preko komandne linije naredbom `python VAS_chess.py`. Nakon toga ispisuje se početno stanje ploče s mogućim pokretima igrača.



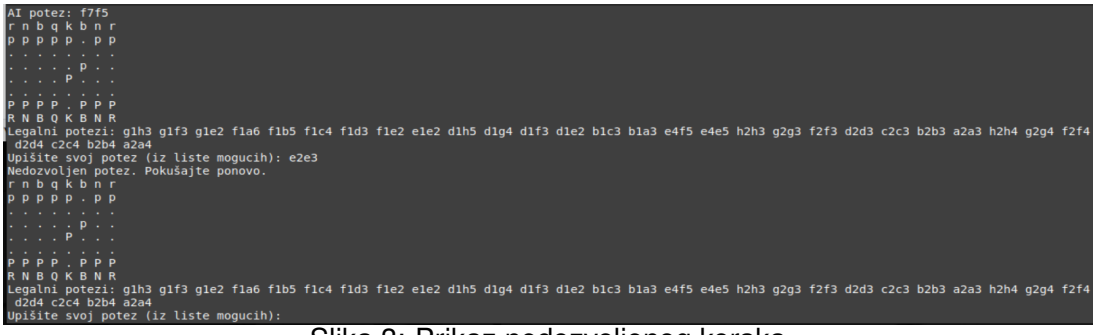
Slika 1: Početak programa

Nakon toga program čeka unos pokreta igrača da se nastavi s radom programa.



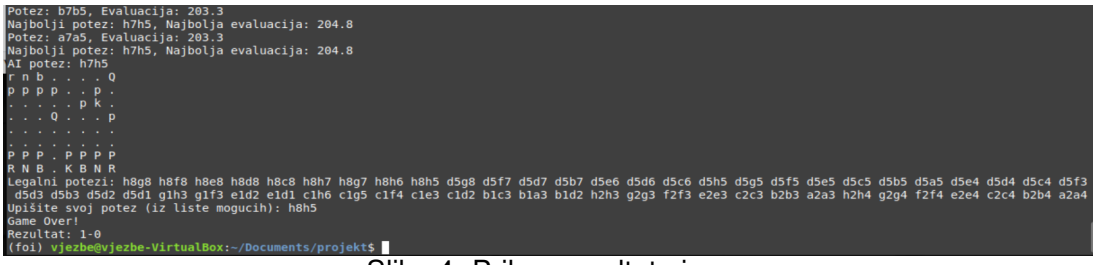
Slika 2: Prvi potez agenta

Nakon što agent izračuna najbolji potez, potez će se izvršiti i ispisati će se korak agenta. Zatim program čeka na ponovan unos igrača.



Slika 3: Prikaz nedozvoljenog koraka

Ako igrač unese nedozvoljen korak, igra ispisuje poruku greške i čeka da igrač unese ispravan potez.



Slika 4: Prikaz rezultata igre

U slučaju šah mata igra završava i prikazuje se rezultat.

3.4. Moguća poboljšanja

Algoritam je dosta jednostavan pa postoji dosta načina na koje bi se mogao poboljšati rad programa. Neki od njih su povečanje dubine pretrage, to je moguće napraviti u mom programu tako da se poveća varijabla `depth = 4`, svakim povećavanjem algoritam ide sve više u dubinu i potrebno je više vremena za svaki korak.

Još neke od mogućih poboljšanja su implementacija alfa-beta rezanje u minimax algoritmu kako bi se smanjio broj evaluacija i ubrzala pretraga. Poboljšanje evaluacije ili dodavanje heuristike koje mogu brže identificirati dobre poteze. Na primjer, može se pridavati veća važnost potezima koji vode do bržeg razvoja figura. Paralelizacija minimax algoritma može znatno ubrzati proces odabira najboljeg poteza. Također je moguće dodati neke od prije poznate taktike kao što su neka od poznatih otvaranja ili razne rošade.[3]

4. Zaključak

U provedbi istraživanja, detaljno se razrađuje Minimax algoritam, njegovi osnovni koncepti te njegova primjena u kontekstu igre šaha. Osim toga, opisuje se izrada simuliranog modela igre šaha koji koristi više agenata, pridržavajući se pravila igre. Eksperimenti su provođeni kako bi se testirale performanse razvijenog sustava, prikupljali podaci o brzini odlučivanja, kvaliteti poteza i reakcijama na različite igračke strategije.

U implementaciji se koristi programski jezik Python, a kao pomoćna biblioteka korištena je `python-chess` za rad sa šahovskim pravilima i potezima.

Zaključno, projekt donosi doprinos razumijevanju i primjeni Minimax algoritma u kontekstu šahovske igre te demonstrira mogućnosti višeagentnih sustava u simulaciji igara. Implementacija omogućuje razvoj inteligentnog protivnika koji može pružiti izazovne i raznolike partije, poboljšavajući iskustvo šahovskih igrača.

Popis literature

[1] „python-chess: a chess library for Python.” Datum pristupa: 07.01.2024. (2024.), adresa: https://python-chess.readthedocs.io/en/latest/#python-chess-a-chess-library-for-python.

[2] G. C. Stockman, „A minimax algorithm better than alpha-beta?” *Artificial Intelligence*, sv. 12, br. 2, str. 179–196, 1979.

[3] Y. Seirawan, *Winning Chess Tactics*. Everyman Chess, 2005.

Popis slika

5. Link na GitHub

Potpuni kod je moguće vidjeti na sljedećoj poveznici:

https://github.com/dodo9715/VAS_projekt_chess