

Veštačka Inteligencija

Izveštaj III faze projekta

Domineering

Naziv tima : **MVP**

Milić Aleksa 17774

Vulić Vladan 16511

Petrović Miloš 16285

Uvod:

Dominacija (Dominnering) je jednostavna matematička strateška igra za dva igrača sa nultom sumom. U Dominaciji dva igrača imaju kolekciju domina (veličine 2×1) koje naizmenično postavljaju na tablu igre, prekrivajući kvadrate. Tabla igre može biti bilo kog oblika kvadrat ili pravougaonik, najčešće se igra na tabli veličine 8×8 . Dva igrača su označena kao Vertikalni i Horizontalni igrači. U standardnoj igri Dominacije prvi igrač je Vertikalni, kome je dozvoljeno samo da svoje domine vertikalno postavlja na tablu a Horizontalni samo horizontalno. Naravno, domine ne smeju da se preklapaju i kao u većini igara u kombinatornoj teoriji igara, prvi igrač koji ne može da postavi dominu gubi igru.

I faza projekta:

U prvoj fazi projekta treba definisati način predstavljanje stanje problema tj igre, osnovne funkcije igre i grafički korisnički interfejs.

Predstavljanje stanje problema (igre):

Trenutno stanje igre se prati u klasi *Game*. U klasi *Game* atribut *matrix* prati stanje igre na tabeli, lista kolona i redova prikazuje slobodne pozicije i zauzete pozicije vertikalnih i horizontalnih domina. Imamo attribute *player1* i *player2* koji su tip klase *Player* kao i atribut *players_turn* koji prati čiji je trenutni potez u igri. Klasa *Player* ima attribute *human_or_pc* koji prikazuje da li je igrač čovek ili računar što će biti relevantno u kasnijim fazama projekta. Takođe sadrži atribut *sign* koja prikazuje oznaku vertikalnog prvog igrača sa simbolom (X) i horizontalnog drugog igrača sa simbolom (O).

class Game:

```
matrix: []
players_turn: Player
player1: Player
player2: Player
```

class Player:

```
def __init__(self, sign, who_plays: bool):
    self.human_or_pc = who_plays # 0-pc, 1-human
    self.sign = sign
```

Funkcija za postavljanje početnog stanja:

Funkciji za postavljanje početnog stanja se prosleđuju vrednosti kolone i vrste table kao i da li je prvi igrač računar ili čovek. Pre funkcije za postavljanje imamo funkcije za unos vrednosti kolona i vrsta table i tip prvog igrača. Postavljaju se vrednosti kolona i vrsta table, inicira se matrica početnog stanja sa praznim poljima. Kreiraju se dva igrača, prvom igraču se dodeljuje simbol X i vrednost da li se radi o računaru ili čoveku a drugom igraču se dodeljuje simbol O i vrednost igrača da se radi o čoveku. Dodelju se atributu *players_turn* prvi igrač jer na početku svake igre prednost ima vertikalni igrač X. Zatim se štampa tabla igre sa početnim stanjima tj korisnički interfejs u konzoli.

```
def __init__(self, human_or_pc1, n: int, m: int):
    self.N = n
    self.M = m
    self.matrix = [{" " for i in range(0, M)] for j in range(0, N)]
    self.player1 = Player("X", human_or_pc1)
    self.player2 = Player("O", True) # 2.player is always human
    self.players_turn = self.player1
    self.print_table()
```

Funkcija koja obezbeđuje prikaz proizvoljnog stanja igre:

Funkcija `print_table()` obezbeđuje prikaz proizvoljnog stanja igre tj table. Funkcija se poziva na kraju funkcije za postavljanje početnog stanja i na kraju funkcije za prelazak u novo stanje `play_a_turn()`. U funkciji se iscrtava matrica stanje igre sa trenutnim vrednostima postavljenih domina u formatiranoj tabeli sa gridom i simbolima vrste i kolona. Ceo prikaz se štampa u konzoli računara.

	A	B	C	D	E	F	
	=	=	=	=	=	=	
6							6
	---	---	---	---	---	---	
5				O	O		5
	---	---	---	---	---	---	
4			X				4
	---	---	---	---	---	---	
3			X				3
	---	---	---	---	---	---	
2							2
	---	---	---	---	---	---	
1							1
	---	---	---	---	---	---	

```
def print_table(self):
    letter = 65 # A
    # vrh table
    print(" ", end=" ") # corner
    for i in range(0, M):
        print(f" {chr(letter + i)}", end=" ")

    print("")
    print(" ", end=" ")
    for i in range(0, M):
        print(" =", end=" ")

    print("")
    # matrix
    for i in range(0, N):
        print(f"{N - i}||", end=" ")
        for j in range(0, M):
            print(f" {self.matrix[i][j]} |", end=" ")
        print(f"{N - i}")
        print(" ", end=" ")
        for _ in range(0, M):
            print(" ---", end=" ")
        print(" ")
```

Funkcija za unos novog stanja:

Funkcija `play_a_turn()` služi za unos novog stanja u igri. Prosleđuje joj se trenutno stanje igre i igrač unosi koordinate table gde želi da postavi dominu. Ukoliko je potez valjan, ažurira se stanje table igre u matrici `matrix` na osnovu čiji je trenutni red igrača `players_turn`. Ukoliko je prvi igrač upisuje se simbol X na zadatu poziciju i poziciju iznad nje ako je drugi igrač upisuje se simbol O na zadatu poziciju i na poziciju desno od nje. Zatim se poziva funkcija `print_table()` za prikaz stanja igre.

```
def play_a_turn(self):
    while True:
        try:
            row = int(input("Unesite vrstu polja: "))
            column = input("Unesite kolonu polja [A-Z]: ")
        except ValueError:
            return False
        else:
            break
    if self.move_valid(row, column):
        m = ord(column) - 65
        n = N - row
        if self.players_turn is self.player1:
            self.matrix[n][m] = 'X'
            self.matrix[n - 1][m] = 'X'
            self.players_turn = self.player2
        else:
            self.matrix[n][m] = 'O'
            self.matrix[n][m + 1] = 'O'
            self.players_turn = self.player1
        self.print_table()
        return True
    else:
        return False
```

Funkcija za proveru da li je potez valjan:

Pre unosa novog stanja prosleđujemo funkciji `move_valid()` stanje igre, poziciju novog unosa igrača na proveru poteza. Na osnovu tipa igrača proveravamo po definisanim pravilima da li je potez van tabele, na ivici tabele ili da li je zadata pozicija zauzeta i vraćamo bool vrednost u odnosu na to da li može da se odigra odgovarajući potez.

```
def move_valid(self, row, column):
    m = ord(column) - 65 # A -> 1
    n = self.N - row # inverted rows
    if self.players_turn is self.player1: # checking if vertical one
        can be placed
        if row < 0 or row >= N or m < 0 or m > M:
            return False
        if self.matrix[n][m] == '' and self.matrix[n - 1][m] == '':
            return True
        else:
            return False
    else: # checking horizontal one
        if row < 0 or row > N or m < 0 or m >= M - 1:
            return False
        if self.matrix[n][m] == '' and self.matrix[n][m + 1] == '':
            return True
        else:
            return False
```

Funkcija za proveru kraj igre:

Na početku svakog poteza funkcija `is_game_over()` proverava da li je igra gotova. Njoj se prosleđuje stanje igre i na osnovu tipa igrača čiji je potez, proverava se stanje da li ima slobodnih vertikalnih ili horizontalnih poteza. Ukoliko ima slobodnih poteza funkcija vraća bool `False` i igra se nastavlja.

```
def is_game_over(self):
    if self.players_turn is self.player1: # any two empty vertical spaces?
        for i in range(0, self.N - 1):
            for j in range(0, self.M):
                if self.matrix[i][j] == '' and self.matrix[i + 1][j] == '':
                    return False
    else: # any two empty horizontal spaces?
        for i in range(0, self.N):
            for j in range(0, self.M - 1):
                if self.matrix[i][j] == '' and self.matrix[i][j + 1] == '':
                    return False
    return True
```

II faza projekta:

U II fazi projekta treba da definišemo operator promena stanja igre. Funkcija koja na osnovu konkretnog poteza menja stanje igre (table) je definisana u I fazi projekta funkcijom za unos novog stanja pomoću funkcije *play_a_turn()*.

Unos početnih parametra igre realizujemo preko sledećeg koda gde unosimo broj vrsta i kolona tabli kao i da li zelimo da igramo protiv računara ili čoveka.

```
while True:
    try:
        N = int(input("Unesite broj vrsta table: "))
        M = int(input("Unesite broj kolona table: "))
    except ValueError:
        print("Nevalidan unos, pokušajte ponovo!")
        continue
    else:
        break
human_or_pc = bool(input("Igrac 1 je X...\nDa li je on covek ili racunar? (0-racunar, 1-covek): "))
```

Unos novog poteza sve dok on nije validan proveravamo ,nakon provere da li je završena igra , sledećim kodom gde u funkciji *play_a_turn()* pozivamo funkciju *move_valid()* koji proverava da li je potez valjan.

```
placed_correctly: bool = game.play_a_turn()
while not placed_correctly:
    print("Nevalidan potez, pokušajte ponovo!")
    placed_correctly: bool = game.play_a_turn()
```

Ukoliko potez nije valjan posleđujemo bool *False* da je nevalidan potez i pozivamo funkciju *play_a_turn()* sve dok ne odigramo validan potez. Nakon sto odigramo validan potez stanje table se menja i štampano novonastalo stanje table. U sledećem potezu pozivamo prvo funkciju za proveru kraja igre i ukoliko funkcija vrati bool *True* imamo kraj igre i štampano ko je pobednik igre na osnovu čiji je potez. Ako nije kraj igre nastavljamo igru.

```
if game.is_game_over():
    print("#####\nKraj igre!")
    print("Pobedio je 2. igrac - O!") if game.players_turn.sign == "X" else print("Pobedio je 1. igrac - X!")
    print("#####")
    break
print("Igrac X je na potezu") if igrac1_na_potezu is True else print("Igrac O je na potezu")
```

Funkcija koja formira novo stanje igre

U klasi *Game* dodali smo novi atribut *matrix_states* koja će da pamti sva stanja igre, od početka do kraja igre. Realizovali smo funkciju *add_new_state()* koja pravi trenutnu kopiju stanja igre tj atributa *matrix* i dodaje u niz stanja *matrix_states* koji se pamte do kraja igre. Funkciju *add_new_state()* pozivamo u okviru funkcije *play_a_turn()* nakon sto promenimo stanje igre.

```
def add_new_state(self):
    #Kopira trenutno stanje table i dodaje u listu stanja
    self.matrix_states.append(copy.deepcopy(self.matrix))
```

Funkcija za formiranje svih mogućih stanja igre

Realizovali smo funkciju `find_all_available_states()` i dodali smo novi atribut `all_available_states`, tipa lista, klasi `Player`. Funkciju pozivamo u okviru funkcije `play_a_turn()` pre zahteva za unos željenog poteza. Funkcija `find_all_available_states()` pre svega prazni listu `all_available_states` ukoliko ima prethodno ubačena stanja nakon toga na osnovnu tipa igrača, parametra i stanja trenutne table proverava svako slobodno mesto i dodaje na to mesto odgovarajuću dominu i snima kopiju stanja table i ubacuje u listu `all_available_states`. Na završetku funkcije generisali smo svako moguće stanje igrača koju on može da odigra i sačuvali smo u njegovoj listi.

```
def find_all_available_states(self):
    self.players_turn.all_available_states.clear()
    if self.players_turn is self.player1: # any two empty vertical spaces?
        for i in range(0, self.N - 1):
            for j in range(0, self.M):
                if self.matrix[i][j] == '' and self.matrix[i + 1][j] == '':
                    self.matrix[i][j] = 'X'
                    self.matrix[i + 1][j] = 'X'
                    self.player1.all_available_states.append(copy.deepcopy(self.matrix))
                    self.matrix[i][j] = ''
                    self.matrix[i + 1][j] = ''
    else:
        for i in range(0, self.N):
            for j in range(0, self.M - 1):
                if self.matrix[i][j] == '' and self.matrix[i][j + 1] == '':
                    self.matrix[i][j] = 'O'
                    self.matrix[i][j + 1] = 'O'
                    self.player2.all_available_states.append(copy.deepcopy(self.matrix))
                    self.matrix[i][j] = ''
                    self.matrix[i][j + 1] = ''
```

Napisali smo i funkciju koja prikazuje sva moguća stanja koja su odigrana u igri ili sva moguća stanja koje igrač može da odigra u njegovom potezu prosleđivanjem odgovarajuće liste stanja.

```
def print_states(self, matrica):
    #Stampa sva dosadasnja stanja u terminalu
    for k in range(0, len(matrica)):
        if(matrica == self.matrix_states):
            print(f"State number {k}")
        else:
            print(f"Available move num. {k}")
    letter = 65 # A
    # vrh table
    print(" ", end=" ") # corner
    for i in range(0, M):
        print(f" {chr(letter + i)}", end=" ")
    print("")
    print(" ", end=" ")
    for i in range(0, M):
        print(" =", end=" ")
    print("")

    for i in range(0, N):
        print(f"{{N - i}}|", end=" ")
        for j in range(a0, M): # counting backwards
            print(f"{{matrica[k][i][j]}} |", end=" ")
        print(f"{{N - i}}")
        print(" ", end=" ")
        for _ in range(0, M):
            print(" ---", end=" ")
        print(" ")
```

```

### Game Domineering ###
Unesite broj vrsta table: 4
Unesite broj kolona table: 4
Igrac 1 je X...
Da li je on covek ili racunar? (0-racunar, 1-covek): 1
Igrac 2 je O
  A  B  C  D
  =  =  =  =
4||  |  |  |  ||4
  ---
3||  |  |  |  ||3
  ---
2||  |  |  |  ||2
  ---
1||  |  |  |  ||1
  ---
Igrac X je na potezu

```

Primer 1: Početak Igre

```

Unesite vrstu polja: 1
Unesite kolonu polja [A-Z]: A
  A  B  C  D
  =  =  =  =
4||  |  |  |  ||4
  ---
3||  |  |  |  ||3
  ---
2|| X |  |  |  ||2
  ---
1|| X |  |  |  ||1
  ---

```

Primer 3: Unos novog stanja

```

Available move num. 0
  A  B  C  D
  =  =  =  =
4|| X |  |  |  ||4
  ---
3|| X |  |  |  ||3
  ---
2||  |  |  |  ||2
  ---
1||  |  |  |  ||1
  ---
Available move num. 1
  A  B  C  D
  =  =  =  =
4||  | X |  |  ||4
  ---
3||  | X |  |  ||3
  ---
2||  |  |  |  ||2
  ---
1||  |  |  |  ||1
  ---
Available move num. 2
  A  B  C  D
  =  =  =  =
4||  |  | X |  ||4
  ---
3||  |  | X |  ||3
  ---
2||  |  |  |  ||2
  ---
1||  |  |  |  ||1
  ---

```

Primer 2: Prikaz
prvi 3 moguća
poteza

```

Unesite vrstu polja: 3
Unesite kolonu polja [A-Z]: A
  A  B  C  D
  =  =  =  =
4|| X | O | O | X ||4
  ---
3|| X | X |  | X ||3
  ---
2|| X | X | O | O ||2
  ---
1|| X |  | O | O ||1
  ---
#####
Kraj igre!
Pobedio je 1. igrac - X!
#####

```

Primer 4: Kraj igre

III faza projekta:

U III fazi projekta potrebno je da implementiramo Min-Max algoritam sa alfa-beta odsecanjem za igru Domineering koja omogućava racunaru da odigra poteze. Ali kako on uopste funkcioniše i sta je on ustvari?

Min-Max algoritam je heurističko pretraživanje stabla koje se koristi u računarskim igrama za određivanje optimalnog poteza. Alfa-beta odsecanje je optimizacija za Min-Max algoritam koja smanjuje broj poteza koje treba proceniti tako što prekida pretraživanje stabla za poteze koje se zna da nisu optimalni.

Primer:

1. Koren stabla: sva moguća stanja igre nakon prvog poteza
2. Prvi nivo stabla: sva moguća stanja igre nakon drugog poteza (našeg protivnika)
3. Drugi nivo stabla: sva moguća stanja igre nakon trećeg poteza (naš potez)
4. Treći nivo stabla: sva moguća stanja igre nakon četvrtog poteza (potez protivnika)
5. Četvrti nivo stabla: sva moguća stanja igre nakon petog poteza (naš potez)
6. Peti nivo stabla: procena stanja igre (pobeda, poraz ili nerešeno)

Potrebno je da ga implementiramo uz odredjena pravila:

- ***Na osnovu zadatog stanja problema***

Ovo je reseno tako što prvo ocenjujemo poteze našeg protivnika (min potez), a zatim ocenjujemo poteze koje možetemo odigrati vi (max potez). Cilj je pronaći potez s najvećom očekivanom dobiti za nas ili najmanjim očekivanim gubitkom.

Na primer, ako se igra Domineering i mi smo na potezu, možemo napraviti stablo poteza koje obuhvata sve moguće poteze koje možetemo odigrati u trenutnom stanju igre. Zatim, za svaki od tih poteza, napravimo podstablo koje obuhvata sve moguće sledeće poteze nakon što se odigra taj potez.

Na primer, kompjuter je na osnovu **trenutnog stanje** igre (problema) zaključio koji potez bi odigrao

Trenutno stanje table									
		A	B	C	D	E			
		=	=	=	=	=			
5		X							5

4		X		0		0			4

3									3

2									2

1									1

Kompjuter je igrao: [0,3]									

Ovo je na osnovu prethodnog stanja odigrao potez [0,3] I dobili smo novo stanje igre

Trenutno stanje table									
		A	B	C	D	E			
		=	=	=	=	=			
5		X				X			5

4		X		0		0		X	4

3									3

2									2

1									1

I tako dalje, algoritam radi **na osnovu zadatog stanja problema**

- ***Na osnovu dubine pretraživanja***

Ovo je reseno tako sto algoritam zadrzi funkciju parametra dubine, odnosno depth. Imajuci u vidu kako radi algoritam, dok on kreira stablo poteza koje obuhvataa sve moguće poteze koje možetemo odigrati u trenutnom stanju igre, ono ipak treba sadrzati neko ogranicenje. Jer bi kreiranje tog stabla poteza moglo da potraje veoma dugo (u zavisnosti od moci racunara).

```
def minmax_pruning(self, depth: int, player: bool, a, b):
    if depth == 0: #cant go any deeper,
```

Prilikom svakog rekurzivnog poziva, smanjujemo dubinu do koje algoritam pravi stablo odlucivanja.

```
self.minmax_pruning(depth - 1, not player, -b, -a)
```

Koje ce se eventualno završiti ogranicenjem if depth == 0 I tada ce se izvršiti sekcija pod sledecim pravilom:

- ***Na osnovu procene stanja (heuristike) koja se određuje kada se dostigne zadata dubina traženja.***

Heuristika je funkcija koja se koristi u Min-Max algoritmu za procenu stanja igre. Ona se obično koristi kada se dostigne zadana dubina pretraživanja stabla i služi za ocenjivanje koliko je stanje igre blizu pobede ili poraza. Kod nas, dubina pretraživanja je zadata na sledeci nacin:

```
Dubina preporuke: Za tablu 4x4 na dubini 8, kompjuter je nepobediv, a za 8x8 na dubini 5, bice razumno brz.
Dubina alphabete: 10
```

Nakon sto se 10 puta rekurzivno pozove funkcija i redom smanjuje dubina pozvace se funkcija heuristike

```
if depth == 0: #cant go any deeper,
    return self.heuristics(player) #
```

A njena implementacija:

```
def heuristics(self, player):
    #that player vs opponent
    return self.get_num_of_ava_placements(player) - self.get_num_of_ava_placements(not player), None, None
```

Ona vraca procenu stanja, odnosno broj mogucih poteza jednog igraca i njegovog protivnika.

- ***Vraća potez koji treba odigrati ili stanje u koje treba preći***

Osim sto heuristika vraća nepostojeće poteze (bilo je neophodno staviti None, None kako bi prijem rezultata funkcije ostao kompatibilan). Nasa funkcija vraća poteze na način:

```
return a, ri if ri >= 0 else self.i, rj if rj >= 0 else self.j
```

Gde a predstavlja alphu, ri – return value od i-tog indexa I rj – return value od j-tog indexa, a self.i i self.j predstavljaju vrednosti indexa koje su bile sacuvane ako bi se vrednosti izgubile tokom rada rekurzije.

Realizovati funkcije koje obezbeđuju odigravanje partije između čoveka i računara:

Funkcija koja ovo obezbeđuje se naziva play_turn i ona je bila definisana u prethodnoj fazi, ali u ovoj, sadrži znatne izmene a to su:

```
else: #else COMPUTER plays
    matrix2 = copy.deepcopy(self.matrix)
    _, row, column = self.minmax_pruning(self.depth, False, -math.inf, math.inf)
    print(f"Kompiuter je igrao: [{row},{column}]" )
    self.matrix = matrix2
    self.set_domino(row, column, False)
    self.players_turn = self.player2

self.add_new_state()
return True
```

Gde omogućujemo postavku domina od strane odabira kompjutera.

```

def minmax_pruning(self, depth: int, player: bool, a, b):
    if depth == 0: #cant go any deeper,
        return self.heuristics(player) #using heuristic to estimate how good t

    ri, rj = -12345, -12345 # random init
    for i in range(self.N):
        for j in range(self.M):
            if self.set_domino(i, j, player): #try to play that position for e
                if not player: # so, placement is ok, if X plays, remember tha
                    self.i, self.j = i, j
                eva, _, _ = self.minmax_pruning(depth - 1, not player, -b, -a)
                eva = - eva
                self.remove_item(i, j, player) #remove placed item, we need to
                if eva > a: # The value of eval is used to update the alpha va
                    a = eva
                    ri, rj = i, j #these are good placements
                    if b <= a: #If the alpha value ever becomes greater than
                        return b, ri, rj #return the evaluation and indexes.

    return a, ri if ri >= 0 else self.i, rj if rj >= 0 else self.j #self.i an

```

Ovom implementacijom (na prethodnoj slici) Obezbedjujemo procenu stanja koja se zasniva na pravilima zakljucivanja. Vidimo da za parametre ima, self – koja se referencira na svoj objekat data klase, dubinom i **IGRACA** za kojeg se racuna valjanost stanja, kao i samo stanje za koje se racuna procena.

Mehanizam koji je koriscen lezi u prethodnom skupu definisanih pravila kojim na ova igra namece i koji implementirani iskorisceni u prethodnim fazama. I konacno vidimo da ova funkcija implementira prevodjenje stanje u listu cinjenica.