

**Izveštaj projekta:**

**Blockade V\_1.0.0**

**Mentori:**

Mr. Vladan Mihajlović

**Studenti:**

**Vukašin Branković 16011**

**Dimitrije Aleksić 15988**

Sadržaj

[PRVA FAZA- princip rada igre 3](#_Toc93231577)

[Zahtevi 4](#_Toc93231578)

[main.py 4](#_Toc93231579)

[game.py 5](#_Toc93231580)

[tabla.py 6](#_Toc93231581)

[igrac.py 8](#_Toc93231582)

[zid.py 9](#_Toc93231583)

[DRUGA FAZA 10](#_Toc93231584)

[TREĆA FAZA 11](#_Toc93231585)

[Minimax 11](#_Toc93231586)

[algorithm.py 11](#_Toc93231587)

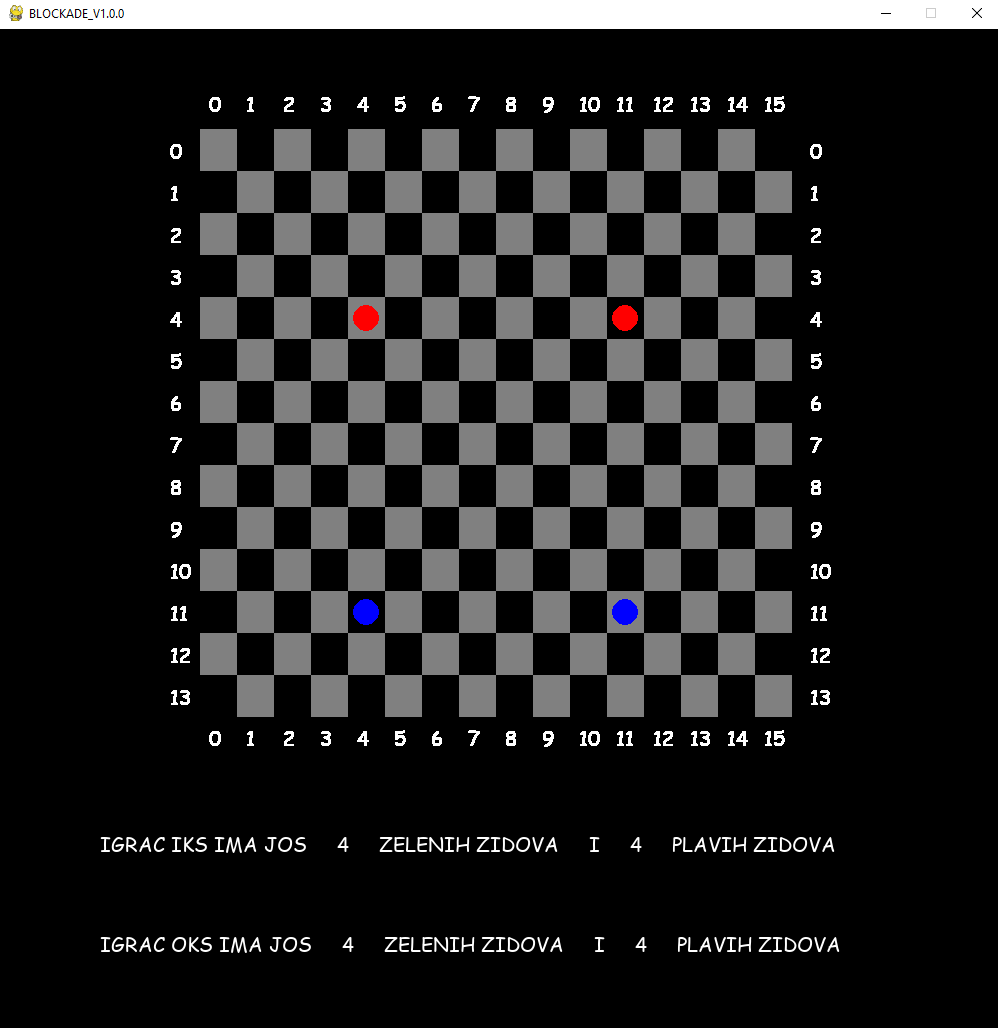
[algorithmZid.py 13](#_Toc93231588)

[Dodaci u fajlu main.py 13](#_Toc93231589)

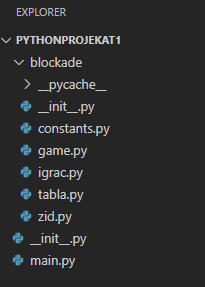
[Dodaci u fajlu game.py 14](#_Toc93231590)

[Dodaci u fajlu tabla.py 14](#_Toc93231591)

[ČETVRTA FAZA 16](#_Toc93231592)

Projekat je realizovan kao 2D igra korišćenjem biblioteke pygame. O načinima korišćenja i primene data biblioteke najviše smo saznali preko youtube kanala „Tech With Tim”. Pygame je omogućio da se interfejs umesto preko konzole iscrtava na sledeći način.

Slika 1.) Grafički izgled interfejsa

Radi lakšeg kodiranja i preglednosti, kod smo pisali u šest različita fajla:

main.py

tabla.py

igrac.py

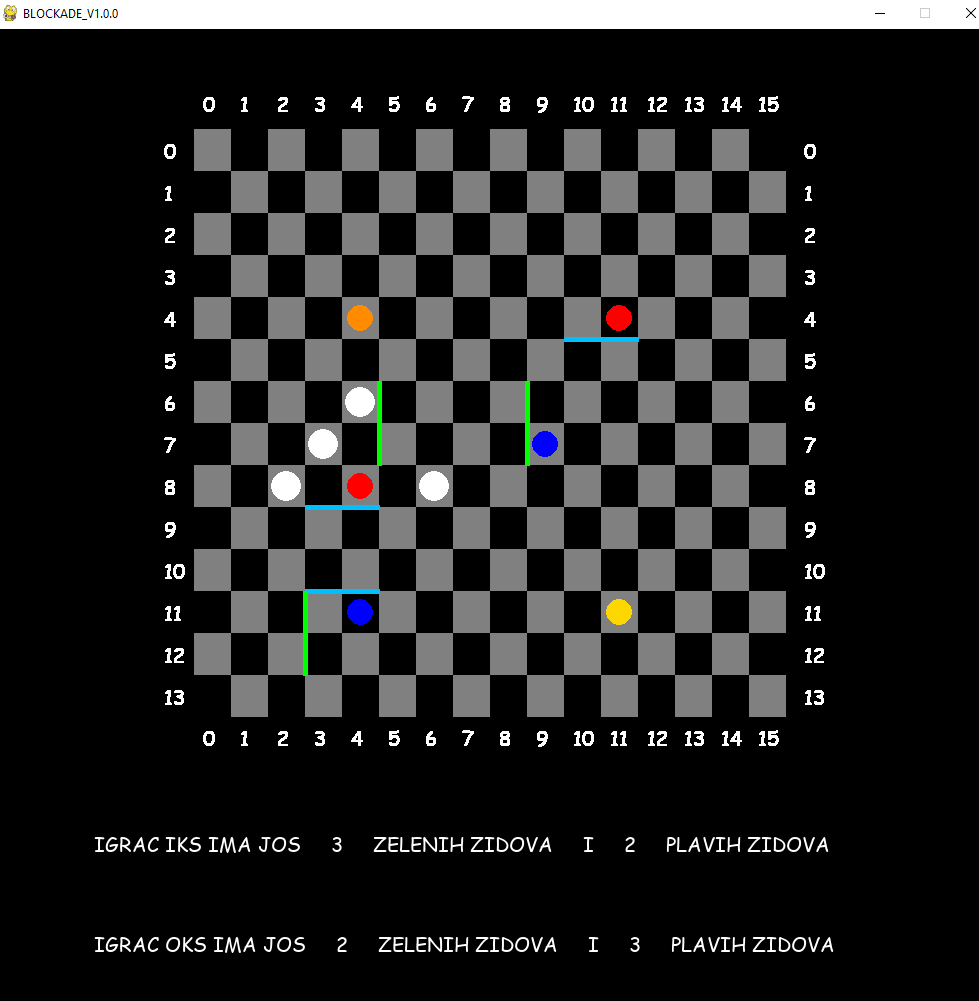
zid.py

game.py

constants.

# PRVA FAZA- princip rada igre

Igru igraju dva čoveka. Pre početka igre se unose početni parametri. Svaki čovek ima po dva igrača na tabli. Prvo igra crveni(IKS) igrač. U svakom potezu je moguće pomeriti igrača i nacrtati zid. Igrači igraju naizmenično. Ukoliko želimo da nacrtamo zeleni(vertikalni) zid, klikom na taster z I pritiskom miša zid se crta na poziciji na kojoj je kliknuto. Za crtanje plavih(horizontalnih) zidova se koristi taster p i klik miša. Takodje radi lakšeg odigravanja partije, implementirano je I iscrtavanje svih validnih poteza koje igrač može da odigra. Izgled korisničkog ekrana nakon par odigranih poteza bi izgleda0 kao na sledećoj slici. Beli krugovi predstavljaju moguće pozicije koje igrač može da odigra. Narandžasti predstavljaju početne pozicije crvenog(IKS) igrača a žute predstavljaju početne pozicije plavog(OKS) igrača.



Slika 2.) Prozor igre nakon par odigranih poteza

## Zahtevi

Uspeli smo da implementiramo svaki zahtev koji se očekivao za prvu fazu. Za svako trenutno stanje, tabla se iscrtava zajedno sa brojevima kolona i vrsta sa strane. Pre početka igre korisnik kroz terminal unosi početne parametre igre(broj vrsta i kolona, početne pozicije igrača i broj zidova). Implementirano je pomeranje igrača i postavljanje zidova klikom mišem. Kada igrač dodje na početno polje protivničkog igrača, ekran će se obojiti u boju pobednika i vršiće se ispisivanje natpisa u kojem stoje čestitke i ko je pobedio. Prvi uvek igra računar. Igrač IKS je crvene boje a igrač OKS plave boje. Funkcije koje proveravaju da li je potez validan ili ne su realizovane na način tako da igrači ne mogu da odigraju nevalidne poteze, nego samo one koji su u skladu sa pravilima igre. To se odnosi na pomeranje pešaka i crtanje zidova. U nastavku možete videti šta je sve implementirano u datoj fazi.

## main.py

Glavna komponenta ovog fajla jeste main metoda u kojoj se nalazi run petlja (beskonacna petlja) u kojoj se poziva metoda .update() nad objektom tipa Game. Cilj metode je da izvrši sva isrtavanja koja korisnik može videti na ekranu. Iscrtavanja se vrše 60 puta u sekundi. To je definisano naredbom clock.tick(FPS). FPS je konstanta i ima vrednost 60.

def main():

    clock = pygame.time.Clock()

    game = Game(WIN)

    run =True

    while run: #run petlja

        game.update()

        clock.tick(FPS)

Prilikom izvršavanja igre stalno dolazi do nekih dogadjaja kao što su: pritisak miša, pritisak tastera, zatvaranja glavnog prozora igre i slično. Pomoću for petlje je implementiran odgovor aplikacije na prva tri dogadjaja koja smo naveli. Ukoliko dodje do zatvaranje glavnog prozora prekida se sa izvršenjem.

for event in pygame.event.get():

            if event.type==pygame.QUIT:

               run=False

            if event.type == pygame.MOUSEBUTTONDOWN:

                pos = pygame.mouse.get\_pos()

                vrsta, kolona = pozicija\_misa(pos)

                game.select(vrsta, kolona)

                if keys\_pressed[pygame.K\_z]:

                  pos = pygame.mouse.get\_pos()

                  vrsta, kolona = pozicija\_misa(pos)

                  game.select2(int(vrsta), int(kolona))

                elif keys\_pressed[pygame.K\_p]:

                   pos = pygame.mouse.get\_pos()

                   vrsta, kolona = pozicija\_misa(pos)

                   game.select3(int(vrsta), int(kolona))

Nakon svakog iscrtavanje I svake iteracije u run petlji neophodno je pozvati sledeću metodu koja vrši update glavnog prozora igre.

 pygame.display.update()

## game.py

Kao atribute, klasa Game sadrži objekat tipa Tabla nad kojom se pozivaju metode za iscrtavanje. Takodje sadži listu valid\_moves u kojoj se kao tuple upisuju pozicije koje je moguće odigrati. Atribut turn odredjuje koji igrač je na redu da igra a atribut selected se koristi kako bi se sprečilo da dodje do iscrtavanja nevalidnih mogućih poteza ukoliko igrač selektuje drugog igrača a već je selektovao jednog.

class Game:

    def \_\_init\_\_(self, win):  #init je inicialising the game

        self.win = win

        self.selected = 0

        self.tabla = Tabla()

        self.turn = RED

        self.zeleni = RED

        self.plavi = RED

        self.valid\_moves = []

Glavna metoda klase Game je .update() koja poziva metode nad objektom Tabla za crtanje svih igraca, zidova, početnih pozicija i mogućih poteza

    def update(self):

        self.tabla.draw\_squares(self.win)

        self.tabla.nacrtaj\_pocetne\_poz(self.win)

        self.tabla.draw(self.win)

        self.draw\_valid\_moves(self.valid\_moves)

        self.proveri\_pobednika()

Metoda select se poziva kada se selektuje igrač na tabli I u njoj se pozivaju metode za iscrtavanje mogućih poteza. Select2 se poziva prilikom kreiranja zelenih zidova I ona vrši upis objekta Zid u listu. Select3 je slična kao Select2 samo se poziva prilikom kreiranja horizontalnih zidova. U klasi Game se takodje nalaze metode za proveru kraja igre(da li je neko od igrača došao na početnu poziciju protivničkog) I ispisa poruke koji igrač je pobedio ukoliko dodje do pobede.

## tabla.py

Klasa Tabla je zadužena za iscrtavanje polja na tabli kao i iscrtavanje brojeva vrsti I kolona na sve četiri strane table. Takodje je zadužena za unos početnih parametara igre pozivanjem metoda za učitavanje. Sadrži metode za iscrtavanje početnih pozicija igrača I proveru koji su potezi validni a koji ne. Od attributa sadrži listu tabla u kojoj se nalazi onoliko listi koliko ima kolona. A u svakoj toj listi se nalazi onoliko listi koliko ima vrsta. Cilj je da svako polje na tabli ima listu u kojoj će biti upisana nula ako je polje prazno ili Igrač() ukoliko se na tom polju nalazi igrač. Na taj način je moguće lako sprovesti pomeranje igrača I ispitivanje na kojim pozicijama je moguće da se prebaci. Popuna liste se odvija metodom .create\_tablu() . Takodje postoje liste zidVertikalni I zidHorizontalni koje se koriste po istom principu kao lista tabla samo su one namenjene za zidove a ne igrače.

lass Tabla:

    def \_\_init\_\_(self)

    self.kolona = 0

        self.vrsta = 0

        self.tabla = []  #tu se pamti gde se svaki igrac nalazi

        self.zidVertikalni = []

        self.zidHorizontalni = []

self.ucitaj\_vrstu\_kolonu()

        self.ucitaj\_igrace()

        self.ucitaj\_zidove()

        print('CESTITAMO, USPESNO STE UNELI POCETNE PARAMETRE, MOZETE POCETI SA IGROM')

        self.pocetni1 = Igrac(IGRAC\_IKS1\_VRSTA, IGRAC\_IKS1\_KOLONA, ORANGE)

        self.pocetni2 = Igrac(IGRAC\_IKS2\_VRSTA, IGRAC\_IKS2\_KOLONA, ORANGE)

        self.pocetni3 = Igrac(IGRAC\_OKS1\_VRSTA, IGRAC\_OKS1\_KOLONA, GOLD)

        self.pocetni4 = Igrac(IGRAC\_OKS2\_VRSTA, IGRAC\_OKS2\_KOLONA, GOLD)

        self.create\_tablu()

        self.kreiraj\_zidove()

Metoda draw\_squares() je zaduzena za isrtavanje sivih kockica na svakoj drugoj poziciji na tabli. Takodje je neophodno iscrtati i crne kockice na ostalim pozicijama zbog pomeranja igrača tokom partije.

def draw\_squares(self, win): #crta crna i siva polja na tabli

        for kolona in range(KOLONA):

            for vrsta in range(VRSTA):

                pygame.draw.rect(win, BLACK, (vrsta\*POLJE\_HEIGHT + 2/10\* HEIGHT, kolona \*POLJE\_WIDTH + 1/10\* WIDTH, POLJE\_HEIGHT, POLJE\_WIDTH ))

        for kolona in range(KOLONA):

            for vrsta in range(kolona % 2, VRSTA, 2):

                pygame.draw.rect(win, GREY, (vrsta\*POLJE\_HEIGHT + 2/10\* HEIGHT, kolona \*POLJE\_WIDTH + 1/10\* WIDTH, POLJE\_HEIGHT, POLJE\_WIDTH ))

Najvažniji deo klase Tabla jeste validacija odnosno ispitivanje koje sve pozicije igrač može da odigra. Znamo da igrač može da odigra različite poteze u zavisnosti od trenutnog stanja na tabli. Sledeća metoda je zadužene za to.

    def get\_valid\_moves(self, igrac):

        moves = []

        #print (self.tabla)

        if igrac.boja == RED or igrac.boja == BLUE:

          self.ispitaj\_dole(igrac, moves)

          self.ispitaj\_gore(igrac, moves)

          self.ispitaj\_levo(igrac, moves)

          self.ispitaj\_desno(igrac, moves)

          self.ispitaj\_dijagonala1(igrac, moves)

          self.ispitaj\_dijagonala2(igrac, moves)

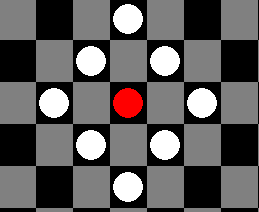
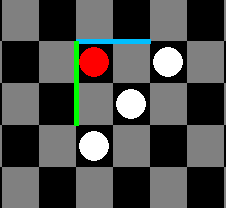
          self.ispitaj\_dijagonala3(igrac, moves)

          self.ispitaj\_dijagonala4(igrac, moves)

          self.ispitaj\_zavrsni\_potez(igrac, moves)

        return moves

Metoda get\_valid\_moves poziva novih 9 metoda u kojoj se ispituje svaki pravac koji igrač može da odigra a to su: gore, dole, levo, desno, I po dijagonali. Implementirano je tako da se obezbedi da igrač može odigrati samo one poteze koji su predvidjeni pravilima igre. Na taj način obezbedjeno je lako i sigurno odigravanje partije bez straha o potencijalnim nevalidnim potezima. Kao rezultat se vraća lista moves u kojoj se kao Tuple nalaze pozicije(vrste i kolone) na koje je moguće pomeriti igrača.



Osim toga sadrži I metodu za pomeranje igrača na tabli tako što u listi menja poziciju gde je igrač trenutno i gde treba da bude.

def move(self, igrac, vrsta, kolona):

        self.tabla[igrac.kolona][igrac.vrsta], self.tabla[kolona][vrsta] = self.tabla[kolona][vrsta], self.tabla[igrac.kolona][igrac.vrsta] #zamena mesta, igrac na pocetnoj poziciji u listi se prebacuje na sledecoj

        igrac.move(vrsta, kolona)

## igrac.py

Klasa igrač kao atribute sadrži vrstu i kolonu pozicije na kojoj se igrač nalazi. Takodje sadrži atribut boja i atribute x i y. Atributi x i y se odredjuju na osnovu vrste i kolone korišćenjem metode calc\_pos(). Oni su bitni zbog toga sto je za isrtavanje kruga i primitiva neophodno da se navedu vrednosti u pixelima. Iscrtavanje se odvija korišćenjem metode draw. Padding odredjuje za koliko je krug uvučen od kraja polja. Takodje klasa igrač sadrži i metode za pomeranje u kojima se vrednosti vrste i kolone zamenjuju novim vrednostima.

def \_\_init\_\_(self, vrsta, kolona, boja, polje\_height, polje\_width):

        self.vrsta = vrsta - 1

        self.kolona = kolona - 1

        self.boja = boja

        self.polje\_height = polje\_height

        self.polje\_width = polje\_width

        self.padding = self.polje\_height/9

        self.x = 0

        self.y = 0

        self.calc\_pos()

    def calc\_pos(self):

        self.x = self.polje\_height \* self.vrsta + 2/10 \* height + self.polje\_height // 2

        self.y = self.polje\_width \* self.kolona + 1/10\* width + self.polje\_width // 2

    def draw(self, win):

        radius = self.polje\_height//2 - self.padding

        pygame.draw.circle(win, self.boja, (self.x, self.y), radius)

## zid.py

Klasa zid je slična klasi Igrač. Obe sadrže atribute: vrsta, kolona, boja, x,y , metodu calc\_pos i metodu za pomeranje pomeri(). Iscrtavanje se vrši jednostavnom narednom pygame.draw.line u kojoj se navode kordinate odakle dokle se linija crta. Optimalna debljina linije (zida) je 5. Crtaju se i vertikalni i horizontalni zidovi. Zamišljeno je da se klikom miša dva puta pozovu metode za crtanje zidova i da se u listi zidova dva polja popune vrednostima Zid() kako bi se lakše sprovelo ispitivanje validnih poteza igrača. Vertikalni zidovi se crtaju na polju na kojem je kliknuto i na jednom polju ispod i iscrtavaju se sa desne strane polja. Dok se horizontalni crtaju na polju na kojem je kliknuto i na jednom polju sa desne strane dok se iscravanje vrši sa donje strane polja

class Zid:

    def \_\_init\_\_(self, vrsta, kolona, boja, polje\_height, polje\_width):

        self.vrsta = vrsta +1

        self.kolona = kolona

        self.boja = boja

        self.polje\_height = polje\_height

        self.polje\_width = polje\_width

        self.x = 0

        self.y = 0

        self.calc\_pos()

    def calc\_pos(self):

        self.x = self.polje\_height \* self.vrsta + 2/10 \* height

        self.y = self.polje\_width \* self.kolona + 1/10\* width

    def drawVertikalni(self, win):

        pygame.draw.line(win, self.boja, (self.x, self.y), (self.x , self.y + self.polje\_width), width=5)

    def drawHorizontalni(self, win):

        pygame.draw.line(win, self.boja, (self.x, self.y), (self.x + self.polje\_height, self.y ), width=5)

    def pomeri(self, vrsta, kolona):

        self.vrsta = vrsta

        self.kolona = kolona

        self.calc\_pos()

# DRUGA FAZA

Situacije na tabli (stanja) prestavljane su korišćenjem tri listi i to su lista tabla, lista zidhorizontalni I lista zidvertikalni. Lista tabla sluzi za pamcenje pozicija igrača, lista zidhorizontalni služi za pamćenje pozicija plavih zidova a lista zidvertikalni za pamćenje pozicija zelenih zidova.Svaka promena na tabli prati I promenu u tim listama. Ako je polje prazno u listi je upisana nula ako nije u listi je upisan objekat tipa Igrac ili Zid. Na osnovu trenutne situacije na tabli i zadatog validnog poteza formira se nova situacija (vrši se iscrtavanje igrača I zidova na novim pozicijama) I takodje se vrše promene u listama na osnovu kojih se obavlja iscrtavanje. Na osnovu trenutne situacije na tabli, klikom miša na igrača. Formiraće se lista svih validnih poteza koje igrač može da odigra To je lista valid\_moves u kojoj se kao tuple nalaze pozicije (vrsta I kolona) koje je moguće odigrati. Validni potezi će se na tabli iscrtavati u obliku belih krugova. Realizovane su sve funkcije koje obezbedjuju odigravanje partije izmedju dva igrača.Unos poteza se vrši klikom miša I nemoguće je izabrati nevalidne poteze. Odigravanjem poteza se vrši promena stanja u listama i I prikaz novonastalog stanja na tabli.Takodje se vrši I provera kraja igre, ukoliko dodje do pobede, ekran će se oboiti u boju pobednika I vrši će se ispis poruke ko je pobedio (plavi ili crveni igrač). Što se tiče funkcije koje proveravaju da li se postavljanjem zida zatvara put do bilo kog početnog polja, to smo pokušali da realizujemo primenom algoritma a zvezda. Ideja je bila da se prilikom postavljanja zida traži put od svakog igrača do njegovih ciljnih pozicija I da ukoliko je put nadjen, onda se dozvoli sa postavljanjem zida a ako nije da se vrši ispis poruke u kojoj se traži da se zid postavi na nekoj drugoj poziciji. Medjutim nismo uspeli da do kraja faze dva napišemo ispravan kod, potrudićemo se da to realizujemo do krajnjeg roka projekta.

# TREĆA FAZA

U ovoj fazi biće definisane i izmene koje su se desile u fazi dva a nisu prethodno navedene u izveštaju. Implementirali smo minimax algoritam koji vraća poteze. Implementirali smo da se ti potezi odigraju (računar ih sam odigrava). Implementirali smo dfs algoritam za pronaženje puta koja sprečava blokiranje puta igrača do njegovih ciljnih pozicija. Implementirali smo metode za izračunavanje heuristike kada se dostigne dubina nula u minimaxu. Izvršili smo ispravke sitnih grešaka koje su postojale. U nastavku možete videti najbitnije izmene i dodatke u odnosu na prethodne faze.

## Minimax

Dodat je folder minimax u kojem se nalaze dva fajla. U tim fajlovima je realizovan algoritam minimax sa alfa beta odsecanjem. Zbog toga što kombinacijom pomeranja igrača i postavljanje zida bi se generisao ogroman broj mogućih stanja i vreme koje bi bilo potrebno računaru da generiše potez bi iznosilo više od predviđenih 30 sekundi. Mi smo implementirali dva minimax algoritma. Prvi vraća tablu sa odigranim potezom igrača dok drugi vraća tablu sa postavljenim zidom. Takodje metode za izračunavanje heuristike bi bile preciznije i bolje jer bi se posebno računala heuristika za pomeranje igrača i posebno za postavljanje zida . Fajlovi u kojima je to realizovano su:

-algorithm.py

-algorithmZid.py

### algorithm.py

Glavna metoda je metoda minimax koja kao parametre uzima trenutno stanje table (objekat Tabla), dubinu, da li je max ili min igrač, figuru odnosno objekat Igrač. Figuru je neophodno sprovesti kako bi se taj objekat kao parametar preneo metodi proceni\_stanje\_igrac koja računa heuristiku za data stanja kada se dostigne dubina nula. Algoritam je realizovan kao na času sa blagim izmenama. Jedna od tih je atribut najbolji\_igrac koji vraća minimax kad je igrač max a koji je neophodan kako bi se kroz konzolu ispisivao potez koji je AI odigrao.

def minimax(tabla, dubina, max\_igrac, figura, alpha, beta):

    global tablica

    if dubina == 0:

        return tabla.proceni\_stanje\_igrac(figura, tablica, brojac), tabla

    if max\_igrac == True:

        maxProcena = float('-inf')

        najbolji\_potez = None

        for potez in vrati\_sve\_poteze(tabla, RED):

            evaluation, desk = minimax(potez[0], dubina-1, False, potez[1], alpha, beta)

            maxProcena = max(maxProcena, evaluation)

            alpha = max(alpha, maxProcena)

            if beta <= alpha:

                break

            if maxProcena == evaluation:

                najbolji\_igrac = potez[1]

                najbolji\_potez = potez

        return maxProcena, najbolji\_potez, najbolji\_igrac

    else:

        minEval = float('inf')

        best\_move = None

        for move in vrati\_sve\_poteze(tabla, BLUE):

            evaluation, desk = minimax(move[0], dubina-1, True, move[1], alpha, beta)

            minEval = min(minEval, evaluation)

            beta = min(beta, minEval)

            if beta <= alpha:

                break

            if minEval == evaluation:

                best\_move = move

        return minEval, best\_move

Realizovana je i metoda vrati\_sve\_poteze koja za dato stanje na tabeli (pozicije igrača) generiše sva moguća stanja, odnosno sve moguće validne pozicije koje igrač može da odigra. Rezultat ove metode je lista moves u kojoj se nalaze sva nova stanja (objekat tipa Tabla u kojoj je odigran validan potez). Takodje se pamti i privremena\_figura kako bi znali koja figura se pomerila na tabli.

Metoda simuliraj\_potez vrši simulaciju odigravanja poteza. Kao atribute dobija figuru koja treba da se pomeri, poziciju na koju se pomera i trenutno stanje table. Jedino što ona radi jeste poziv metode .move nad tablom koja figuru na odredjenoj poziciji u listi pomera na novu poziciju. Kao rezultat, vraća novu tablu sa novim stanjem na njoj.

Metoda da\_li\_je\_bio\_potez vrši ispitivanje da li je odredjeni potez već odigran kako igrač ne bi stalno odigravao jedne iste poteze.U listi tablica se upisuju svi potezi koji su odigrani i njihova cena. Cilj je da već odigran potez ima veću heuristiku od neodigranog kako bi forsirali AI da odabere pravi put.

### algorithmZid.py

Skoro potpuno ista implementacija kao za algorithm.py sa tom izmenom da se poziva druga metoda za računanje heuristike kada se dostigne dubina nula. To je metoda proceni\_stanje\_zid. Takodje, kako računar ne bi vraćao nevalidne poteze koji bi doveli do zatvaranje nekog puta od igrača do cilja, potezi koji imaju trenutnu najbolju heuristiku za njih se vrši ispitivanje da li zatvaraju put. To je ostvareno tako kako bi se uštedelo procesorsko vreme I kako se nebi za svaki potez vršilo dato ispitivanje. Već samo za one koji su najbolji. Ispitivanje se vrši metodom ne\_zatvara\_put koja poziva metodu pronađi\_put. Ukoliko sve metode vrate True, to znači da postoji put I potez je validan

## Dodaci u fajlu main.py

Računar igra kao crveni igrač i igra prvi. Dva puta se vrši pozivanje minimax algoritma. Prvi vraća tablu sa odigranim potezom dok drugi vraća tablu sa postavljenim zidom. Nakon poziva ovih metoda poziva se metoda ai\_potez koja novu tablu dodaje u objekat tipa Game. Takodje nakon svakog poteza igrača vrši se ispit pozicija koje je igrač odigrao i na koje je postavio zid.

if game.turn == RED:

            vrednost, nova\_tabla, figura = minimax(game.tabla, 3 , True, None, float('-inf'), float('inf'))

            game.ai\_potez(nova\_tabla)

            prethodni\_potez(nova\_tabla[1], 0)

            vrednost, nova\_tabla2, zid = minimax2(game.tabla, 1 , True, game, None, None, float('-inf'), float('inf'))

            if nova\_tabla2 != None:

                game.ai\_potez\_zid(nova\_tabla2)

            else:

                game.ai\_potez(nova\_tabla)

                game.promeni\_potez()

            print('AI je pomerio pesaka na polje: ', 'VRSTA' ,(figura.kolona+1) ,'KOLONA' , (figura.vrsta+1) ,'I POSTAVIO JE ZID NA', 'VRSTA', zid[0]+1, 'KOLONA' ,zid[1]+1)

## Dodaci u fajlu game.py

Kao što smo već rekli, dodate su dve metode. Metoda ai\_potez i metoda ai\_potez\_zid. Cilj ovih metoda je da postave novu tablu kao trenutno stanje igre. Nakon dodavanje zida vrši se promena poteza (RED -> BLUE)

def ai\_potez(self, tabla):

        self.tabla = tabla[0]

        self.update()

    def ai\_potez\_zid(self, tabla):

        self.tabla = tabla

        self.promeni\_potez()

        self.update()

Dodata je metoda proveri\_da\_li\_zid\_zatvara() koja prilikom postavljanja zida proverava da li će zid zablokirati neki put do cilja (Od bilo kog igrača do bilo kog ciljnog polja). To je ostvareno preko DFS algoritma. Ukoliko dfs vrati True to znači da put postoji i da je moguće postaviti zid. Ako vrati False vrši će se ispit poruke koja kaže da je to nevalidan potez i biće sprečeno postavljanje zida. Takodje izvršena je ispravka validacije u select2 i select3 metodama koje su zadužene za postavljanje zida. Ispravljene su greške.

## Dodaci u fajlu tabla.py

Ovde je došlo do najvećeg broja izmena i dodavanje novih funkcionalnosti. Ispravljene su greške i nedostaci u metodi get\_valid\_moves(). Dodate su metode sve\_kombinacije\_zidova\_plavih() i sve\_kombinacije\_zidova\_zelenih(). Ove metode za trenutno stanje table vrše pronalaženje svih mogućih pozicija na koje je moguće postaviti zid. Kao rezultat ovih funkcija dobija se lista u kojoj se kao tuple nalaze pozicije (kolona, vrsta) na koje je moguće postaviti validan zid. Validacija je slična kao za metode select 2 i select3 u klasi Game, kada igrač odigrava postavljanje zida.

def sve\_kombinacije\_zidova\_zelenih(self):

        kombinacije\_zelenih = []

        for kolona in range(self.kolona-1):

            for vrsta in range (self.vrsta-1):

                if kolona +1 < self.kolona:

                    if self.zidVertikalni[kolona][vrsta] == 0 and self.zidVertikalni[kolona+1][vrsta] == 0:

                        if self.zidHorizontalni[kolona][vrsta] != 0 and self.zidHorizontalni[kolona][vrsta+1] != 0:

                            pass

                        else:

                            kombinacije\_zelenih.append((kolona,vrsta))

        return(kombinacije\_zelenih)

Dodata je metoda proceni\_stanje\_igrač() za izračunavanje heuristike stanja za odigravanje poteza igrača kada se dostigne dubina nula u minimax algoritmu. Logika za izračunavanje je ta da je najbolji potez onaj koji će biti najbliži nekom od ciljnih stanja. Takodje je implementirano da prethodni potezi imaju veću cenu kako se AI ne bi vrteo u krug. Najbolji način za izračunavanje bi možda bio kada bi se računao broj skokova izmedju igrača I ciljnog stanja, I onaj sa najnižim brojem skokova bi bio najbolji potez. To bi se ostvarilo primenom A zvezda algoritma. Nadamo se da ćemo uspeti to da implementiramo do kraja faze4.

Metoda proceni\_stanje\_zid() se takođe koristi za izračunavanje heuristike ali za minimax2 algoritam kad se dostigne dubina nula kada se vrši postavljanje zida. Logika je ta da su zeleni zidovi(vertikalni) skuplji od plavih i AI bi trebalo da prvo njih forsira zbog toga što se ciljno polje nalazi sa leve ili desne strane. Stanja u kojima je zid na najbližoj poziciji protivničkom igraču imaju najbolju heuristiku i ona se biraju. Glavni cilj je bio da se spreči ili bar oteža pobeda protivničkog igrača konstantim blokiranjem zidovima. Takodje je obezbedjeno da AI ne vrati nevalidne poteze i da ne zablokira bilo koji put.

Dodate su metode dodaj\_plavi\_zid() I dodaj\_zeleni\_zid() koja u već postojeće liste vrši upis zidova na prosleđenim pozicijama.

Glavni dodatak je metoda pronađi\_putDFS() koja koristi DFS algoritam za pronalaženje puta. Metoda vraća True ako je put pronađen i False ako nije. U drugoj fazi smo probali da put pronalazimo preko a zvezda međutim nismo uspeli pa smo u ovoj fazi koristili DFS kao lakšu opciju. Za dodavanje suseda tekućeg čvora se koristi već implementirana metoda get\_valid\_moves(). Na taj način put se traži preko pozicija koje igrač može da odigra. Metoda pronađi\_put() vrši 8 puta poziv DFS metode kako bi se pronašao put za sve igrače do svih ciljnih pozicija. Ukoliko svih 8 vrate True to znači da je potez validan.

# ČETVRTA FAZA

U želji da vreme odigravanja partija bude što kraće i procena heuristike bude što bolja, napravili smo da dva minimax algoritma i dve metode za procenu heuristike. Prvi vraća najbolji mogući potez igrača dok drugi vraća najbolju poziciju na kojoj je moguće postaviti zid. Logika je ta da najbolji potez za pomeranje igrača je onaj koji će igrača dovesti što bliže cilju a najbolje mesto za postavljanje zida je ono koje će dovesti do toga da se protivničkom igraču znazno oteža dolazak do cilja. Metoda za računanje heuristike za pomeranje igrača je proceni\_stanje\_igrac().

def proceni\_stanje\_igrac(self, figura, tablica\_prethodnih\_stanja):

        global tablica

        kolona = figura.kolona

        vrsta = figura.vrsta

        for index , stanje in enumerate(tablica\_prethodnih\_stanja[-5:]):

            if tablica\_prethodnih\_stanja == None:

                pass

            elif stanje[0].vrsta == vrsta and stanje[0].kolona == kolona:

                cena = stanje[1]

                cena-=500

                tablica[index] = (stanje[0], cena)

                return -abs(abs(kolona - self.pocetni4.kolona) + abs(vrsta - self.pocetni4.vrsta)) - abs(cena)

        procena1 = -abs(abs(kolona - self.pocetni3.kolona) + abs(vrsta - self.pocetni3.vrsta))

        procena2 = -abs(abs(kolona - self.pocetni4.kolona) + abs(vrsta - self.pocetni4.vrsta))

        if procena1 > procena2:

            return procena1

        else:

            return procena2

Ova metoda kao parametre uzima figuru (objekat tipa Igrač) koji se nalazi na novoj poziciji u novom stanju i za koje se računa rastojanje do nekog od odrešnih polja. Zbog toga što je ovakav način izračunavanja heuristike dovodio do toga da se igrač vrti u krug (bira jedne iste poteze) uveli smo tablicu\_prethodnih\_poteza. To je lista u kojoj se nalaze pozicije koje je igrač prethodno odigrao i cena tih pozicija Svaki put kad računar odigra stanje koje je već odigrao, cena će se povećati. I ukoliko je pozicija igrača u novom stanju jednaka poziciji na kojoj je već bio, to stanje će imati znatno veću cenu, time će AI biti forsiran da bira nove puteve koji će ga dovesti do cilja. Računa se broj kockica između trenutne pozicije igrača i krajnjeg polja i taj broj kockica se vraća kao procena stanja. Stavljena je negativna apsolutna vrednost -abs() zbog toga što minimax bira stanje sa najvećom cenom a u slučaju broja kockica, najbolje stanje ima najmanju cenu i zbog toga se stavlja negativna vrednost. Takodje vrši se procena za oba krajnja polja, i onaj sa najboljom cenom se vraća kroz return.

Za izračunavanje cene stanja za postavljene zidove koristi se metoda proceni\_stanje\_zid(). Ona slično kao prethodni algoritam kao parametre uzima poziciju na kojoj se nalazi novi zid i uzima boju tog zida. Znamo već da su zeleni zidovi vertikalni a plavi horizontalni. Na početku obilazi sva polja u tabli i nalazi PLAVE protivničke igrače. Uzima vrednosti njihovih kolona i vrsta.Zeleni zidovi imaju prednost zbog toga što najbolje blokiraju protivničkog igrača. Zbog toga će stanja kod kojih se zeleni zidovi nalazi ispred protivničkog igrača imati veću cenu od plavog zida na istoj poziciji. Znači što se zid nalazi bliže protivničkom igraču on će imati veću cenu i zeleni zidovi imaju prednost u odnosu na plave.

def proceni\_stanje\_zid(self, pozicija, boja):

        kolona = pozicija[0]

        vrsta = pozicija [1]

        suma = float('-inf')

        for row in self.tabla:

            for igrac in row:

                if igrac != 0 and igrac.boja == BLUE:

                    igrac\_kolona = igrac.kolona

                    igrac\_vrsta = igrac.vrsta-1

                    if boja == GREEN:

                        if igrac.vrsta == vrsta+1:

                            suma1 = -abs(abs(kolona - igrac\_kolona) + abs(vrsta - igrac\_vrsta)) + 0.1

                            suma = max(suma1, suma)

                        else:

                            suma1 = -abs(abs(kolona - igrac\_kolona) + abs(vrsta - igrac\_vrsta))

                            suma = max(suma1, suma)

                    else:

                        suma1 = -abs(abs(kolona - igrac\_kolona) + abs(vrsta - igrac\_vrsta))

                        suma = max(suma1, suma)

        return suma

Obezbedjeno je da AI ne vraća nevalidne poteze zidova i da ne blokira puteve do ciljnih polja zidom.

Konačni izgled table nakon što je računar odigrao potez izgleda ovako.

