Slaganje (Byte)

1. Izveštaj, 28.11.2023.

Za unos početnih parametara igre se koristi funkcija `UnesiParametreIgre()`. Funkcija traži unos veličine table sve dok se ne unese validna veličina ili se tri puta unese nevalidna veličina(u tom slučaju je podrazumevana veličina table 8x8). Validne veličine table su 8x8, 10x10 i 16x16 jer su to jedine veličine koje ispunjavaju uslove da je n parno i da je broj figura na tabli deljiv sa 8.

```
def UnesiParametreIgre():
    ispravanunos = False
    i = 0
    while not ispravanunos and i<3:
        n = input("Uneti dimenziju table N: (N moze biti 8, 10 ili
                  16):")
        if n=='8' or n=='10' or n=='16':
            n= int(n)
            ispravanunos=True
        else:
            print("Neispravan unos.")
        i+=1
    if i==3:
        n=8
        print("Dimenzija table je 8x8")
    return n
```

Inicijalno stanje problema pravi funkcija `startState(n)` koja vraća stanje na početku igre na tabli veličine nxn.

```
def startState(n):
    stanje={'tabela':dict(),'rowNames':list(),'stekovi':{'X':0,'O':0},
            'naPotezu':'X','covek':'X','n':n }
    stanje['rowNames']=initializeRowNames(n)
    tabela={}
    player={
        0:'X',
        1:'O'
    }
```

```
    for i in range(1,n+1):
        tabela[i]={}
        tmp=i%2
        for j in range(1 if tmp==1 else 2,n+1,2):
            tabela[i][j]=[]
            if i!=1 and i!=n:
                tabela[i][j].append(player[tmp])
    stanje['tabela']=tabela
    return stanje
```

Stanje igre je predstavljeno na sledeći način:

```
stanje={
      'tabela':dict(),
      'rowNames':list(),
      'stekovi':{'X':0,'O':0},
      'naPotezu':'X',
      'covek':'X',
      'n':n
}
```

Gde je **tabela** rečnik u kome je ključ vrsta table (1=A, 2=B,...,8=H), a vrednost rečnik - čiji su ključevi kolone, a vrednosti liste sa figurama koje se nalaze na polju u toj vrsti i koloni. Prva figura u listi je figura na dnu steka. Primer izgleda tabele za tablu 8x8:

```
tabela={
            1:{1:[],3:[],5:[],7:[]},
            2:{2:['X'],4:['X'],6:['X'],8:['X']},
            3:{1:['O'],3:['O'],5:['O'],7:['O']},
            4:{2:['X'],4:['X'],6:['X'],8:['X']},
            5:{1:['O'],3:['O'],5:['O'],7:['O']},
            6:{2:['X'],4:['X'],6:['X'],8:['X']},
            7:{1:['O'],3:['O'],5:['O'],7:['O']},
            8:{2:[],4:[],6:[],8:[]}
 }
```

Rečnik **stekovi** sadrži broj stekova koji su u vlasništvu igrača X i igrača O, inicijalno 0 za oba.
naPotezu označava igrača koji je u datom trenutku na potezu, inicijalno je X na potezu.
covek predstavlja slovo kojim je čovek izabrao da igra.
rowNames je pomoćni rečnik koji mapira vrednosti (1=>A, 2=>B,...,8=>H) za potrebe štampanja tabele. rowNames se inicijalizuje pomoćnom funkcijom `initializeRowNames(n)`
n pamti veličinu tabele.

Za izbor ko će igrati prvi, koristi se funkcija `koPrviIgra()` koja traži unos znaka kojim će igrati čovek tri puta. U slučaju neispravnog unosa sva tri puta, čovek igra kao X. Funkcija vraća izabrani znak.

```
def koPrviIgra():
    var = 'X'
    ispravanunos = False
    i = 0
    while not ispravanunos and i<3:
        value = input("Zelite da igrate kao prvi igrac(X) ili drugi
igrac(O)? Uneti odgovarajuci karakter:")
        if value == 'x' or value =='X':
            ispravanunos=True
        else:
            if value == 'o' or value == 'O':
                var = 'O'
                ispravanunos=True
            else:
                print('Molimo unesite X ili O')
        i+=1
    if i==3:
        print("Igrate kao prvi igrac")

    return var
```

Prikaz stanja igre se vrši funkcijom `showState(stanje)`

```
def showState(stanje):

    def polje_za_stampu(i,j,indeks):
        return ' ' if j not in stanje['tabela'][i] else '.' if
      len(stanje['tabela'][i][j])<indeks+1 else
      stanje['tabela'][i][j][indeks]
    def printText(txt):
        print(txt,end="")

    polje=" {prvo}{drugo}{trece}"

    for i in range(1,stanje['n']+1):
        if i >=10:
            printText("  {}".format(i))
        else:
```

```
            printText("   {}".format(i))
    print()

    for i in range(1,stanje['n']+1):
        for redStampe in range(0,3):
            if redStampe!=1:
                printText(" ")
            else:
                printText(stanje['rowNames'][i])
                  for j in range(1,stanje['n']+1):
                       printText(polje.format(
                       prvo=polje_za_stampu( i,j,6-(3*redStampe)),
                       drugo=polje_za_stampu( i,j,6-(3*redStampe)+1),
                       trece=polje_za_stampu( i,j,6-(3*redStampe)+2)))
                 print()
   print("X:{} O:{}".format(stanje['stekovi']['X'],
   stanje['stekovi']['O']))
```

Unos poteza se obavlja u funkciji `UnesiPotez(stanje)` koja traži unos poteza sve dok se ne unese ispravan potez. Potez se unosi kao string u formatu “{red} {kolona} {pozicija na steku} {smer kretanja}”

```
def UnesiPotez(stanje):
    ispravno = False
    i = 1
    while not ispravno:
        red = input("{} na potezu! Unesi
potez:".format(stanje['naPotezu']))
        potez = red.split(' ')

        if(len(potez)==4 and len(potez[0])==1 and potez[2].isdigit()
and potez[1].isdigit()):
            potez[2]=int(potez[2])
            potez[0] = ord(potez[0]) - ord('A') + 1
            potez[1] = int(potez[1])
            ispravno = ispravanPotez(potez[0],potez[1],potez[2],
potez[3],stanje) and valjanostPoteza(potez[0],potez[1],potez[2],
potez[3],stanje)

        if i%4==0:
            print("Uputstvo za unos poteza - bitno!!!")
```

```
            print("Molimo unesite svoje poteze u sledecem formatu: RED
            KOLONA POZICIJA_NA_STEKU GL/GD/DL/DD")
            print("GL - Gore Levo. GD - Gore Desno. DL - Dole Levo. DD
            - Dole Desno")
        i+=1
    return potez
```

Linija `potez[0] = ord(potez[0]) - ord('A') + 1` pretvara red unesen u obliku karaktera u odgovarajući broj reda kako je reprezentovan u tabeli.

Provera ispravnosti poteza se vrši funkcijom `ispravanPotez(red,kolona,mesto_na_steku,smer_pomeranja,stanje)` koja vraća True ako u tabeli iz stanja postoje uneti red i kolona, i na tom polju na unetom mestu postoji figura i to od igrača koji je trenutno na potezu, i ako je smer pomeranja jedan od 4 moguća.

```
def ispravanPotez(red,kolona,mesto_na_steku,smer_pomeranja,stanje):
    tabela=stanje['tabela']

    if(red in tabela
    and kolona in tabela[red]
    and mesto_na_steku>=0
    and mesto_na_steku<len(tabela[red][kolona])
    and tabela[red][kolona][mesto_na_steku]==stanje['naPotezu']
    and smer_pomeranja in smerovi_kretanja
    ):
        return True
    return False
```

`smerovi_kretanja` je tuple koji sadrži dozvoljene smerove kretanja - `('GL','GD','DL','DD')`

Za proveru da li je kraj igre postoji funkcija `KrajIgre(stanje)` koja proverava da li neko od igrača ima više od polovine stekova ili je tabla prazna.

```
def KrajIgre(stanje):
    velicinatable=stanje['n']
    stekovi=stanje['stekovi']
    tabla=stanje['tabela']
    kraj_igre=False
    granica = velicinatable*(velicinatable-2)//32

    if stekovi['X']>granica:
        kraj_igre=True
    else:
```

```
        if stekovi['O']>granica:
            kraj_igre=True
        else:
            if((stekovi['O']+stekovi['X'])==
            (velicinatable*(velicinatable-2)/16)):
                kraj_igre=True
    return kraj_igre
```

Igra je završena kada je tabla prazna ili kada jedan od igrača ima više od polovine stekova.
Objašnjenje za granicu:
Figure se postavljaju u sve sem prvog i zadnjeg reda(velicinatable-2) ali u svakom redu u svaku drugu kolonu(velicinatable/2), pa je ukupan broj figura na tabli velicinatable*(velicina-2)/2. Da bi se dobio broj stekova, ukupan broj figura se deli sa 8, jer jedan stek sadrži 8 figura, pa je granica velicinatable*(velicina-2)/(2*8*2)=velicinatable*(velicina-2)/32

Kada je tabla prazna, svaki stek će biti u vlasništvu nekog od igrača, pa će zbir stekova X i O biti jednak ukupnom broju stekova koji se mogu formirati, pa je provera `(stekovi['O']+stekovi['X'])==(velicinatable*(velicinatable-2)/16)` zapravo provera da li je tabla prazna

2. Izveštaj, 7.12.2023.

Provera popunjenosti susednih polja obavlja se funkcijom

```
def okolinaPrazna(red, kolona,stanje):
    return len(list(filter(lambda p:
      len(stanje['tabela'][p[0]][p[1]])!=0,
      validniPomeraji(red,kolona,stanje['n']))))==0
```

Povratna vrednost funkcije `validniPomeraji(red,kolona,n)`, koja će biti objašnjena u nastavku, se filtrira samo na pomeraje koji vode do nepraznih pozicija unutar tabele stanja. Ako je dužina liste nepraznih pomeraja jednaka nuli znači da su svi pomeraji vodili do praznih pozicija (vratiće se True ako je okolina prazna, i False u suprotnom)

Na osnovu konkretnog poteza i stanja igre proverava se da li on vodi ka jednom od najbližih stekova:

```
def priblizavanjeNajblizemSteku(red,kolona,smer_pomeranja,stanje):
    najblizi=najbliziStekovi(red,kolona,stanje)
    pomeraj=pomeraji[smer_pomeranja]
    prvobitna_razdaljina=distance((red,kolona),najblizi[0])
    razdaljine=[distance((red+pomeraj[0],kolona+pomeraj[1]),polje) for
      polje in najblizi]
    return len(list(filter(lambda x:x<prvobitna_razdaljina,
      razdaljine)))!=0
```

Proverava da li se stek pomeranjem sa pozicije `(red,kolona)` u smeru `smer_pomeranja` približava jednom od najbližih stekova. Najbliže stekove dobijamo pozivom istoimene funkcije.

Funkcija `najbliziStekovi(red,kolona,stanje)` vratiće listu pozicija stekova.
U promenljivoj `prvobitna_razdaljina` pamti se početna razdaljina između datog polja i najbližeg steka, dok se u promenljivoj `razdaljine` pamti lista razdaljina između svih susednih pozicija nakon primene pomeraja i pozicija na najbližem steku.
Povratna vrednost funkcije predstavlja rezultat provere postojanja vrednosti iz liste `razdaljine` koja je manja od `prvobitna_razdaljina`.

Izgled funkcije `najbliziStekovi(red,kolona,stanje)`:

```
def najbliziStekovi(red,kolona,stanje):
    tabela=stanje['tabela']
    n=stanje['n']
    najblizi=list()
    obidjeni=list()
    obidjeni.append((red,kolona,))
    neobidjeni=list(map(lambda
      x:(x[0],x[1]),validniPomeraji(red,kolona,n)))
    while(len(najblizi)==0 and len(neobidjeni)!=0):
```

```
        nove_destinacije=list()
        for polje in neobidjeni:
            obidjeni.append(polje)
            if len(tabela[polje[0]][polje[1]])!=0:
                najblizi.append(polje)
            destinacije=list(map(lambda x:(x[0],x[1]),
             validniPomeraji(polje[0],polje[1],n)))
            for dest in destinacije:
                if dest not in obidjeni and dest not in neobidjeni and
                 dest not in nove_destinacije:
                    nove_destinacije.append(dest)
        neobidjeni=nove_destinacije
    return najblizi
```

Po principu obilaska po širini počinje se od početne pozicije i prate susedne dobijene iz funkcije `validniPomeraji(red,kolona,n)`. Osim liste obiđenih i neobiđenih, koristi se i lista `nove_destinacije` koja čuva destinacije koje će se obraditi u narednom koraku. Kad se naiđe na polja koja sarže stekove, ona se stavljaju u listu `najblizi` nakon čega se pretraga obustavlja.

Na osnovu konkretnog poteza i stanja igre funkcija `valjanostPoteza`:

```
def valjanostPoteza(red, kolona,mesto_na_steku,smer_pomeranja,stanje):
novo_polje=(red+pomeraji[smer_pomeranja][0],kolona+pomeraji[smer_pomera
nja][1],smer_pomeranja)
if novo_polje not in validniPomeraji(red,kolona,stanje['n']):
      return False

broj_figura_na_novom_polju=len(stanje['tabela'][novo_polje[0]][novo_pol
je[1]])
if(broj_figura_na_novom_polju==0):
      if mesto_na_steku!=0:
            return False
      if not okolinaPrazna(red,kolona,stanje):
            return False
      return priblizavanjeNajblizemSteku( red, kolona, smer_pomeranja,
             stanje)

if not (mesto_na_steku)<broj_figura_na_novom_polju:
      return False
if not (broj_figura_na_novom_polju+len(stanje['tabela'][red][kolona])-
   mesto_na_steku)<=8:
      return False
return True
```

proverava da li se potez može odigrati prema pravilima pomeranja.

Kreira se nova pozicija na osnovu trenutne i smera pomeranja `novo_polje`. Nakon provere validnosti pozicije, broje se figure na tom mestu.
Ako nema figura, nizom kratkih provera (da li se pomera ceo stek, da li je okolina prazna, da li se približava najbližem steku) utvrđuje se da li je potez valjan.
Ako ima figura, proverava se da li je moguće pomeriti traženi broj figura.
Ako sve provere prođu funkcija vraća True.

Već pomenuta funkcija `validniPomeraji(red,kolona,n)` glasi:

```
def validniPomeraji(red,kolona,n):
moguciPomeraji=[(red+pomeraji[smer][0],kolona+pomeraji[smer][1],smer)
for smer in smerovi_kretanja]
    return list(filter(lambda p:p[0]>=1 and p[0]<=n and p[1]>=1 and
            p[1]<=n,moguciPomeraji))
```

jednostavno generiše listu mogućih pomeraja i zadržava samo one pomeraje koji su unutar granica kvadratne table.

Menjanje trenutnog stanja na osnovu konkretnog poteza postiže se funkcijom `OdigrajPotez(potez,original)`:

```
def OdigrajPotez(potez,original):
    red = potez[0]
    vrsta = potez[1]
    novi_red = red + pomeraji[potez[3]][0]
    nova_vrsta = vrsta + pomeraji[potez[3]][1]
    mesto_u_steku = potez[2]

    stanje=marshal.loads(marshal.dumps(original))
    premesta_se = stanje["tabela"][red][vrsta][mesto_u_steku:]
    stanje["tabela"][red][vrsta] =
stanje["tabela"][red][vrsta][:mesto_u_steku]
    stanje["tabela"][novi_red][nova_vrsta].extend(premesta_se)

    if len(stanje["tabela"][novi_red][nova_vrsta]) == 8:
        stanje["stekovi"][stanje["tabela"][novi_red][nova_vrsta][-1]]
+= 1
        stanje["tabela"][novi_red][nova_vrsta].clear()

    return stanje
```

koja na osnovu unetog poteza određuje trenutnu poziciju na tabli i poziciju na koju će se prebaciti figure. Sa `marshal.loads(marshal.dumps(original))` se pravi duboka kopija početnog stanja kako bi se izbeglo nepoželjno deljenje referenci. U promenljivu `premesta_se` se izdvajaju figure koje će biti premeštene, i nakon toga se uklanjaju iz

pređašnjeg steka čije su bile deo. Izdvojene figure se dodaju na vrh ciljanog steka sa `stanje["tabela"][novi_red][nova_vrsta].extend(premesta_se)`.

Ako novokreirani stek sadrži 8 figura, stek se prazni i dodaje se poen igraču čija figura je bila na vrhu.

Funkcija koja obezbeđuje samo odigravanje partije glasi:

```
def OdigravanjePartije():
    stanje=startState(UnesiParametreIgre())
    stanje['covek']=koPrviIgra()
    showState(stanje)
    print("Uputstvo za unos poteza - bitno!!!")
    print("Molimo unesite svoje poteze u sledecem formatu: RED KOLONA
           POZICIJA_NA_STEKU GL/GD/DL/DD")
    print("GL - Gore Levo. GD - Gore Desno. DL - Dole Levo. DD - Dole
           Desno")
     while not KrajIgre(stanje):
        potez = UnesiPotez(stanje)
        stanje= OdigrajPotez(potez,stanje)
        if stanje['naPotezu']=='X':
            moguci=moguciPotezi(stanje,'O')
            prvi=next(moguci,None)
            if prvi is not None:
                stanje['naPotezu']='O'
        else:
            moguci=moguciPotezi(stanje,'X')
            prvi=next(moguci,None)
            if prvi is not None:
                stanje['naPotezu']='X'
        showState(stanje)
    saigrac=''
    if stanje['covek']=='X':
        saigrac='O'
    else:
        saigrac='X'

    if stanje['stekovi'][stanje['covek']] > stanje['stekovi'][saigrac]:
        print("Pobedili ste!")
    else:
        if stanje['stekovi'][stanje['covek']] ==
           stanje['stekovi'][saigrac]:
            print("Nereseno!")
```

```
        else:
            print("Izgubili ste!")
    print("Krajnji rezultat je:")
    print("X:{} O:{}".format(stanje['stekovi']['X'],
stanje['stekovi']['O']))
```

Koristi funkcije `startState(n)`, `UnesiParametreIgre()` i `koPrviIgra()` kako bi postavila početne parametre igre. Nakon kratkog uputstva o načinu igre ulazi se u petlju koja se vrti sve dok ne dođe do kraja igre. Potez se unosi i odigrava sa `UnesiPotez(stanje)` i `OdigrajPotez(potez,original)`.

Nakon unošenja poteza proveravamo uz pomoć funkcije `moguciPotezi(stanje,'O')` proveravamo da li suparnik ima moguće poteze. Ako ima, suparnik biva na potezu. Ako nema, prvobitni igrač ostaje na potezu sve dok suparnik ne dobije mogućnost da igra ili dok se igra ne završi.

Koristeći `showState(stanje)` se nakon svakog poteza prikazuje stanje na tabli. Po završetku igre proglašava se pobednik ili da je nerešeno.

Na osnovu zadatog igrača na potezu i zadatog stanja table formiraju se svi mogući potezi:

```
def moguciPotezi(stanje,igrac):
    polja=[(red,kolona,p) for red,v in stanje['tabela'].items() for
kolona,p in v.items()]
    poljaSaFiguramaIgraca =list(filter(lambda x:igrac in x[2],polja))
    figure=[(red,kolona,[m  for m in range(0,len(l)) if l[m]==igrac])
for red,kolona,l in poljaSaFiguramaIgraca]

    for red,kolona,l in figure:
        if okolinaPrazna(red,kolona,stanje):
            if 0 in l:
                najblizi=najbliziStekovi(red,kolona,stanje)
                prvobitna_razdaljina=distance((red,kolona),najblizi[0])
                for pomeraj in validniPomeraji(red,kolona,stanje['n']):
                    razdaljine=[distance((pomeraj[0],pomeraj[1]),polje)
                    for polje in najblizi]
                    if len(list(filter(lambda x:x<prvobitna_razdaljina,
                    razdaljine)))!=0:
                        yield (red,kolona,0,pomeraj[2])
        else:
            for pomeraj in validniPomeraji(red,kolona,stanje['n']):
                polje=(pomeraj[0],pomeraj[1])
```

```
                brojFiguraNoviStek = len(
                  stanje['tabela'][polje[0]][polje[1]])
                dozvoljenoNaStek=8-brojFiguraNoviStek
                najniziDozvoljen=len(stanje['tabela'][red][kolona])
                -dozvoljenoNaStek
                good=list(filter(lambda x:x>=najniziDozvoljen and
                     x<brojFiguraNoviStek,l))
                for m in good:
                     yield (red,kolona,m,pomeraj[2])
```

U listi `polja` se pamte sva polja na tabli sa informacijama o redu, koloni i prisutnim figurama. Lista se filtrira i u listi `poljaSaFiguramaIgraca` se pamte samo polja sa figurama trenutnog igrača. Na osnovu `poljaSaFiguramaIgraca` se generiše lista `figure` koja sadrži red, kolonu i indeks na steku za svaku figuru.
Za svaku figuru proverava se da li može da se izvrši potez:

- Ako je okolina prazna proverava se približavanje najbližem steku i vracamo sa `yield` potez koji je valjan
- Ako okolina nije prazna proverava se mogućnost pomeranja figura sa trenutne pozicije na susedne, proverava se potez za svaku susednu poziciju i svako mesto na steku i vraća sa `yield`

Na osnovu svih mogućih poteza formiraju se sva moguća stanja:

```
def mogucaStanja(stanje,igrac):
    for potez in moguciPotezi(stanje,igrac):
        yield (OdigrajPotez(potez,stanje),potez)
```

Za dobijanje novih stanja koristi se funkcija `OdigrajPotez(potez,original)`.

```
def distance(polje1,polje2):
    return max(abs(polje1[0]-polje2[0]),abs(polje1[1]-polje2[1]))
```

Vraća udaljenost dva polja, koju računa kao maksimum horizontalne i vertikalne udaljenosti.

3. Izveštaj, 29.12.2023.

Igra počinje pozivom funkcije `OdigravanjePartijeSaRacunarom()` čije telo je prikazano:

```
def OdigravanjePartijeSaRacunarom():
    stanje=startState(UnesiParametreIgre())
    stanje['covek']=koPrviIgra()

    showState(stanje)
    print("Uputstvo za unos poteza - bitno!!!")
    print("Molimo unesite svoje poteze u sledecem formatu: RED KOLONA
POZICIJA_NA_STEKU GL/GD/DL/DD")
    print("GL - Gore Levo. GD - Gore Desno. DL - Dole Levo. DD - Dole
Desno")
    brPoteza=0

    maxDubina=2
    while not KrajIgre(stanje):


        if stanje['naPotezu']==stanje['covek']:
            potez = UnesiPotez(stanje)
            stanje= OdigrajPotez(potez,stanje)
        else:
            print("AI na potezu...")

            potezRacunara = iterative_deepening(stanje,maxDubina,True)

            potez=potezRacunara[0]
            maxDubina=potezRacunara[1]
            stanje=OdigrajPotez(potez,stanje)
            brPoteza+=1
            print("{0} {1} {2}
{3}".format(stanje['rowNames'][potez[0]],potez[1],potez[2], potez[3]))

        if stanje['naPotezu']=='X':
            moguci=moguciPotezi(stanje,'O')
            prvi=next(moguci,None)
            if prvi is not None:
                stanje['naPotezu']='O'
        else:
            moguci=moguciPotezi(stanje,'X')
            prvi=next(moguci,None)
            if prvi is not None:
```

```
                stanje['naPotezu']='X'
        showState(stanje)
    saigrac=''
    if stanje['covek']=='X':
        saigrac='O'
    else:
        saigrac='X'

    if stanje['stekovi'][stanje['covek']] > stanje['stekovi'][saigrac]:
        print("Pobedili ste!")
    else:
        if stanje['stekovi'][stanje['covek']] ==
stanje['stekovi'][saigrac]:
            print("Nereseno!")
        else:
            print("Izgubili ste!")
    print("Krajnji rezultat je:")
    print("X:{} O:{}".format(stanje['stekovi']['X'],
stanje['stekovi']['O']))
```

U promenljivoj potezRacunara pamti se povratna vrednost iz funkcije `iterative_deepening(stanje,start_depth,igraRacunar)` nakon čega se partija odvija uobičajenim tokom.

```
def iterative_deepening(stanje,start_depth,igraRacunar):
    start_time=time.time()
    allowed_time=5
    def max_value(prvi,stanje, dubina, alpha, beta, potez=None):
        if not prvi and time.time()-start_time>allowed_time:
            return None
        mogucaStanjaGen = mogucaStanja(stanje,'X' if
stanje['covek']=='O' else 'O')
        moguceStanje=next(mogucaStanjaGen,None)
        if dubina == 0 or KrajIgre(stanje):
            return (potez, proceni_stanje(stanje,'X' if
stanje['covek']=='O' else 'O'),)
        if moguceStanje is None:
            return min_value(prvi,stanje,dubina-1,alpha,beta,potez)
        else:
            while moguceStanje is not None:
                m=min_value(prvi,moguceStanje[0], dubina - 1,alpha,
beta,  moguceStanje[1] if potez is None else potez)
                if m is None:
                    return None
```

```
                alpha = max(alpha, m, key=lambda x: x[1])
                if alpha[1] >= beta[1]:
                    return beta
                moguceStanje=next(mogucaStanjaGen,None)
        return alpha
    def min_value(prvi,stanje, dubina, alpha, beta, potez=None):
        if not prvi and time.time()-start_time>allowed_time:
            return None
        mogucaStanjaGen = mogucaStanja(stanje,stanje['covek'])
        moguceStanje=next(mogucaStanjaGen,None)
        if dubina == 0 or KrajIgre(stanje):
            return (potez, proceni_stanje(stanje,'X' if
stanje['covek']=='O' else 'O'),)
        if moguceStanje is None:
            return max_value(prvi,stanje,dubina-1,alpha,beta,potez)
        else:
            while moguceStanje is not None:
                m=max_value(prvi,moguceStanje[0], dubina - 1,alpha,
beta, moguceStanje[1] if potez is None else potez)
                if m is None:
                    return None
                beta = min(beta, m, key=lambda x: x[1])
                if beta[1] <= alpha[1]:
                    return alpha
                moguceStanje=next(mogucaStanjaGen,None)

        return beta

    def
minimax_alfa_beta(prvi,stanje,dubina,igraRacunar,alpha=(None,-math.inf)
,beta=(None,math.inf)):
        if igraRacunar:
            return max_value(prvi,stanje,dubina,alpha,beta)
        else:
            return min_value(prvi,stanje,dubina,alpha,beta)
    depth=start_depth
    potez=minimax_alfa_beta(True,stanje,depth,igraRacunar)
    depth+=1
    while time.time()-start_time<=allowed_time and depth<30:
        novi_potez=minimax_alfa_beta(False,stanje,depth,igraRacunar)
        if novi_potez:
            potez=novi_potez
            depth+=1
```

```
    return (potez[0],depth-1)
```

Ovaj kod implementira algoritam iterativnog produbljivanja za pretraživanje prostora stanja igre. Očigledno je da je reč o implementaciji Minimax algoritma s alfa-beta odsecanjem, ali s dodatkom iterativnog produbljivanja kako bi se omogućilo pretraživanje u ograničenom vremenskom okviru.

Počinje s minimalnom dubinom `start_depth` i povećava je iterativno.
`max_value(prvi,stanje, dubina, alpha, beta, potez=None)` je funkcija za maksimizaciju vrednosti u Minimax algoritmu. Prima trenutno stanje, dubinu pretrage, alfa i beta vrednosti, potez i parametar 'prvi' - na osnovu ovog parametra se određuje da li će funkcija da gleda vremensko ograničenje ili ne. Ako je parametar prvi *False*, funkcija će vršiti proveru da li je od početka poziva funkcije `iterative_deepening` prošlo više od određenog vremenskog ograničenja - ako jeste, funkcija će vratiti *None*. Ako je parametar prvi True, neće proveravati vremensko ograničenje. Odnosno vraća potez i procenjenu vrednost.

`min_value(prvi,stanje, dubina, alpha, beta, potez=None)` je funkcija za minimizaciju vrednosti u Minimax algoritmu. Prima iste parametre kao i `max_value` i takođe vraća potez i procenjenu vrednost.
I unutar `max_value`i unutar `min_value` funkcije, koristi se generator `mogucaStanjaGen` koji vraća sledeće moguće stanje sve dok ono postoji - petlja obrađuje stanja sve dok se ne dođe do kraja mogućih stanja ili do odsecanja
Ako dubina postane 0 ili je kraj igre onda se vraća procena aktuelnog stanja pozivom funkcije `proceni_stanje(stanje, racunar)` koja će biti kasnije objašnjena.

`potez=minimax_alfa_beta(True,stanje,depth,igraRacunar)` traži potez za početnu dubinu, bez provere vremenskog ograničenja. Ovim se obezbeđuje da funkcija `iterative_deepening` uvek vratiti neki potez, čak i ako za pretragu za početnu dubinu treba duže od postavljenog ograničenja.

`while time.time()-start_time<=allowed_time and depth<30` pokreće petlju u kojoj se sa `if novi_potez` proverava da li je povratna vrednost `minimax_alfa_beta(False,stanje,depth,igraRacunar)` funkcije neprazna, tj. da li je uspela izračunati potez u datom vremenskom ograničenju. Ako je potez uspešno izračunat, ažurira vrednost poteza. Povećava se dubina pretrage za jedan kako bi se nastavilo sa sledećom iteracijom.

`iterative_deepening` vratiće najbolji pronađen potez, kao i poslednju dubinu koja je uspešno pretražena, kako bi se ta dubina iskoristila kao početna dubina za sledeći poziv funkcije.

Kod funkcije `proceni_stanje(stanje, racunar)` izgleda ovako:

```
def proceni_stanje(stanje, racunar):
    if KrajIgre(stanje):
        if
stanje['stekovi'][racunar]>stanje['stekovi'][stanje['covek']]:
```

```
            return 100000000
        else:
            if
stanje['stekovi'][racunar]<stanje['stekovi'][stanje['covek']]:
                return -100000000
            else:
                return 0

    praznaokolina={'X':0,'O':0}
    samojedan={'X':0,'O':0}
    vise={'X':0,'O':0}
    vrhovi={'X':0,'O':0}
    for i in stanje['tabela']:
        for j in stanje['tabela'][i]:
            if len(stanje['tabela'][i][j])>0:
                vrhovi[stanje['tabela'][i][j][-1]]+=1
                pomeraji=validniPomeraji(i,j,stanje['n'])
                zauzeta=0
                for pom in pomeraji:
                    if len(stanje['tabela'][pom[0]][pom[1]])>0:
                        zauzeta+=1
                if(zauzeta==0):

                    praznaokolina[stanje['tabela'][i][j][0]]+=1
                else:
                    if zauzeta==1:
                        samojedan[stanje['tabela'][i][j][0]]+=1
                    else:
                        vise[stanje['tabela'][i][j][0]]+=1

    tezinaPoen=14
    tezinaPrazno=15
    tezinaJedan=5
    tezinaVise=3
    tezinaVrhovi=1

ocenaRacunar=stanje['stekovi'][racunar]*tezinaPoen+praznaokolina[racuna
r]*tezinaPrazno+samojedan[racunar]*tezinaJedan+vise[racunar]*tezinaVise
+vrhovi[racunar]*tezinaVrhovi


ocenaCovek=stanje['stekovi'][stanje['covek']]*tezinaPoen+praznaokolina[
```

```
stanje['covek']]*tezinaPrazno+samojedan[stanje['covek']]*tezinaJedan+vi
se[stanje['covek']]*tezinaVise+vrhovi[stanje['covek']]*tezinaVrhovi
    ocena = ocenaRacunar-ocenaCovek
    return ocena
```

Prvo se proverava da li je igra završena. Ako jeste, procenjuje se pobednik. Ako je računar pobedio, vraća se velika pozitivna vrednost (100000000), ako je čovek pobedio, vraća se velika negativna vrednost (-100000000), a u slučaju nerešenog rezultata, vraća se 0.

Inače, inicijalizuju se četiri rečnika koji će se koristiti za brojanje različitih situacija na tabli u vezi sa figurama na dnu stekova sa praznom okolinom, figurama na dnu stekova sa samo jednim stekom u okolini, figurama na dnu stekova sa više stekova u okolini i vrhovima stekova posebno za oba igrača(`praznaokolina`, `samojedan`, `vise`, `vrhovi`). Iterira se kroz celu tablu i broje se pomenute situacije. Postavljene su težine za svaki od ovih faktora. Ideja sa ovom heuristikom i datim težinama je bazirana na strategiji kontrolisanja što većeg broja figura na dnu slobodnih stekova (stekova bez drugih stekova u okolini). Igrač koji pri kraju igre ima veći broj ovakvih figura će imati veći izbor poteza i veću kontrolu nad redosledom spajanja stekova.
Računaju se ocene za računar i čoveka koristeći vrednosti upamćenih u rečnicima pomnoženih sa njihovim težinama.
Razlika između ocene računara i ocene čoveka se koristi kao konačna ocena stanja koja se vraća kao povratna vrednost funkcije.