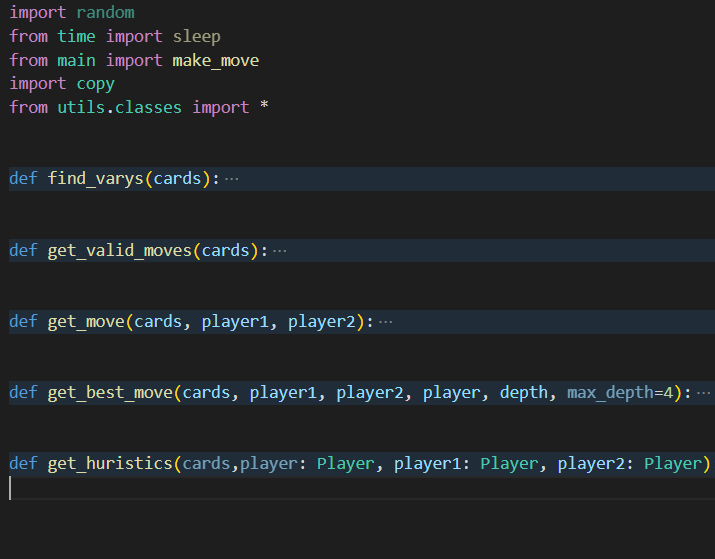
گزارش پروژه فاز اول

# اعضای تیم: بردیا کریمی نیا ، باربد کلیایی

# مقدمه:

در این بخش ابتدا یک توضیح خلاصه از عملکرد فایل mminimax\_agent.py گفته می شود. در وهله اول به دنبال پیدا کردن یک عامل با روش minimax می باشیم که بتواند عامل random\_agent را ببرد. خوب است اشاره کنیم که عامل random\_agent می تواند کاملا تصادفی بازی کند و همواره احتمالی وجود دارد که عامل minimax\_agrnt از آن ببازد. پس خوب است که نتایج بازی را طی چندین بازی مشخص کنیم.(به علت اینکه Random\_agent مقدار seed مشخصی را دریافت نمی کند تا همواره از یک جا شروع کند.)

# Minimax agent:



تابع find\_varys موقعیت واریس را در هر مرحله مشخص می کند.

تابع get\_valid\_moves مکان های قابل حرکت واریس را در اختیار ما می گذارد.

تابع get\_move تابع اصلی برای خروجی دادن حرکتی که عامل می خواهدانجام دهد می باشد.

تابع get\_best\_move درواقع منطق اصلی minimax را در خود جای داده که در هر بازی تا حداکثر عمق 4 پیش می رود تا بتواند بازی را ببرد.

تابع get\_heuristics درواقع هیوریستیک و وزن مورد نظر در یک state از بازی را به ما بر می گرداند.

از آنجایی که توابع find\_varys ,get\_valid\_moves درست مثل توابع عامل random\_agent می باشد پس اط توضیح بیش از آن صرف نظر می کنیم.

# تابع get\_moves:

def get\_move(*cards*, *player1*, *player2*):

num\_cards = len(*cards*)

max\_depth = 4

if num\_cards < 25:

max\_depth = 5

elif num\_cards < 20:

max\_depth = 20

elif num\_cards < 16:

max\_depth = 100

val, best\_move = get\_best\_move(

*cards*, *player1*, *player2*, *player*=*player1*, *depth*=0, *max\_depth*=max\_depth)

return best\_move

در اینجا با استفاده از تابعget\_best\_move می اییم و بهترین همراه با وزن آن را (که به وزن آن احتایج نداریم) را خروجی می دهیم.

ورودی آن state هه کارت ها و player1 , 2 و depth فعلی که می خواهید از آن شروع به انجام الگوریتم minimax کنید را می دهید.

علاوه بر ان ایده دیگری هم پیاده می‌کنیم و ان تغییر عمق دینامیک بر اساس تعداد کارد های باقی مانده است. ما می‌خواهیم سرچمان عمیقترین ممکن باشد ولی در شرایط اولیه بیشتر از ۴ امکان پذیر نیست. اما زمانی که تعداد راس ها کمتر شود عملا می‌توان کل درخت را گشت.

# تابع get\_best\_moves:

def get\_best\_move(*cards*, *player1*, *player2*, *player*, *depth*, *max\_depth*):

if *depth* > *max\_depth*:

return (

get\_huristics(

*cards*=*cards*,

*player*=*player*,

*player1*=*player1*,

*player2*=*player2*,

*ended*=False,

),

None,

)

if *player* == *player1*:

# maximizer player

ans = -1e8

best\_move = None

valid\_moves = get\_valid\_moves(*cards*)

if len(valid\_moves) == 0:

return get\_huristics(

*cards*=*cards*,

*player*=*player*,

*player1*=*player1*,

*player2*=*player2*,

*ended*=True,

), None

for move in valid\_moves:

temp\_cards = copy.deepcopy(*cards*)

make\_move(*cards*=temp\_cards, *move*=move, *player*=*player*, *other\_player*=*player2*)

h\_move, \_ = get\_best\_move(

*cards*=temp\_cards,

*player1*=*player1*,

*player2*=*player2*,

*player*=*player2*,

*depth*=*depth* + 1,

*max\_depth*=*max\_depth*,

)

del temp\_cards

if ans < h\_move:

ans, best\_move = h\_move, move

return ans, best\_move

else:

# minimizer player

ans = 1e8

best\_move = None

valid\_moves = get\_valid\_moves(*cards*)

if len(valid\_moves) == 0:

return get\_huristics(

*cards*=*cards*,

*player*=*player*,

*player1*=*player1*,

*player2*=*player2*,

*ended*=True,

), None

for move in valid\_moves:

temp\_cards = copy.deepcopy(*cards*)

make\_move(*cards*=temp\_cards, *move*=move, *player*=*player*, *other\_player*=*player1*)

h\_move, \_ = get\_best\_move(

*cards*=temp\_cards,

*player1*=*player1*,

*player2*=*player2*,

*player*=*player1*,

*depth*=*depth* + 1,

*max\_depth*=*max\_depth*,

)

del temp\_cards

if ans > h\_move:

ans, best\_move = h\_move, move

return ans, best\_move

ورودی در این تابع:

* cards: state هه کارت ها
* player1: maximizer
* player2: minimizer
* player: player فعلی در این depth
* depth: عمقی که در آن هستیم
* Max\_depth: بیشترین عمقی که الگوریتم در آن انجام می شود.

خروجی تابع :

* Heuristic در node فعلی و بهترین حرکت بعد از آن

در وهله اول اگر عمق از 4 بیشتر بود باید صرفا heuristic های آن state را بررای کارت ها خروجی دهیم و قاعدتا best\_move یی هم در این حالت موجود نیست و None خروجی می دهیم.

حال اگر player هه فعلی maximizer بود باید حررکتی را انجام دهیمکه بیشترین heuristic را به ما بدهد.

پس باید ابتدا حررکات مجاز را دریافت کنیم و روی آن iterate کنیم.

در مرحله بعد ما ابتدا داریمکه چون در هر node وضعیت کارت ها عوض می شوند پس یک copy از آن ها ایجاد می کنیم و روی آن کار می کنیم ولی باید حواسمان به manage کردن حافظه باشد که اگر عمق بالا برود segment fault نخوریم. برای این هم بعد از انجام کار روی کپی از کارت ها آن را پاک می کنیم.

تابع make\_moves برایمان stateهه کارت ها و player وmove را گرفته و آن اعمال می کند.

بعد از آن دوباه باید تابع get\_best\_moves را صدا زده منتهی باید روی player حریف و depth +1 و کارتهای کپی این کار انجام شود.

در نهایت هم باید حرکت و heuristic ای را انتخاب کنیم که بیشترین مقدار را دارد. و آن دو را خروجی دهیم.

برای حالتی که player ما minimizer باشد هم مراحل باز همین می شود ولی باید حرکت و heuristic ای انتخاب شود که کمترین مقدار را داشته باشد.

در کنار اینها اگر دیگر حرکتی نتوان کرد به hue خبر می‌دهیم که برنده ان لحظه را به ما برگرداند به جای حدسش.

# تابع get\_heuristics:

def get\_huristics(*cards*, *player*: Player, *player1*: Player, *player2*: Player, *ended*):

"""

finding the huristics for the given situation and the player

"""

# {'Stark': [8], 'Greyjoy': [7], 'Lannister': [6], 'Targaryen': [5], 'Baratheon': [4], 'Tyrell': [3], 'Tully': [2]}

# for player 1

Stark1 = len(*player1*.cards["Stark"])

Greyjoy1 = len(*player1*.cards["Greyjoy"])

Lannister1 = len(*player1*.cards["Lannister"])

Targaryen1 = len(*player1*.cards["Targaryen"])

Baratheon1 = len(*player1*.cards["Baratheon"])

Tyrell1 = len(*player1*.cards["Tyrell"])

Tully1 = len(*player1*.cards["Tully"])

# for player 2

Stark2 = len(*player2*.cards["Stark"])

Greyjoy2 = len(*player2*.cards["Greyjoy"])

Lannister2 = len(*player2*.cards["Lannister"])

Targaryen2 = len(*player2*.cards["Targaryen"])

Baratheon2 = len(*player2*.cards["Baratheon"])

Tyrell2 = len(*player2*.cards["Tyrell"])

Tully2 = len(*player2*.cards["Tully"])

stark\_sum = Stark1 + Stark2

greyjoy\_sum = Greyjoy1 + Greyjoy2

lannister\_sum = Lannister1 + Lannister2

targaryen\_sum = Targaryen1 + Targaryen2

baratheon\_sum = Baratheon1 + Baratheon2

tyrell\_sum = Tyrell1 + Tyrell2

tully\_sum = Tully1 + Tully2

p1score = 0

p2score = 0

win\_points = 10

diif\_mull = 1.2

if *ended* == True:

if Stark1 > Stark2 or (Stark1 == Stark2 and *player1*.last["Stark"] == 1):

p1score += win\_points

elif Stark2 > Stark1 or (Stark2 == Stark2 and *player2*.last["Stark"] == 1):

p2score += win\_points

if Greyjoy1 > Greyjoy2 or (Greyjoy1 == Greyjoy2 and *player1*.last["Greyjoy"] == 1):

p1score += win\_points

elif Greyjoy2 > Greyjoy1 or (Greyjoy2 == Greyjoy1 and *player2*.last["Greyjoy"] == 1):

p2score += win\_points

if Lannister1 > Lannister2 or (Lannister1 == Lannister2 and *player1*.last["Lannister"] == 1):

p1score += win\_points

elif Lannister2 > Lannister1 or (Lannister2 == Lannister1 and *player2*.last["Lannister"] == 1):

p2score += win\_points

if Targaryen1 > Targaryen2 or (Targaryen1 == Targaryen2 and *player1*.last["Targaryen"] == 1):

p1score += win\_points

elif Targaryen2 > Targaryen1 or (Targaryen2 == Targaryen1 and *player2*.last["Targaryen"] == 1):

p2score += win\_points

if Baratheon1 > Baratheon2 or (Baratheon1 == Baratheon2 and *player1*.last["Baratheon"] == 1):

p1score += win\_points

elif Baratheon2 > Baratheon1 or (Baratheon2 == Baratheon1 and *player2*.last["Baratheon"] == 1):

p2score += win\_points

if Tyrell1 > Tyrell2 or (Tyrell1 == Tyrell2 and *player1*.last["Tyrell"] == 1):

p1score += win\_points

elif Tyrell2 > Tyrell1 or (Tyrell2 == Tyrell1 and *player2*.last["Tyrell"] == 1):

p2score += win\_points

if Tully1 > Tully2 or (Tully1 == Tully2 and *player1*.last["Tully"] == 1):

p1score += win\_points

elif Tully2 > Tully1 or (Tully2 == Tully1 and *player2*.last["Tully"] == 1):

p2score += win\_points

if p1score == p2score and *player1*.last["Stark"] == 1:

p1score += 1

elif p1score == p2score and *player2*.last["Stark"] == 1:

p2score += 1

# if p1score > p2score:

# print(p1score - p2score)

if p1score > p2score:

return 80*;*

else:

return -80*;*

def f(*num*):

return (*num* \* (*num* + 1)) / 2

# tully hue

if tully\_sum == 2:

if *player1*.last["Tully"] == 1:

p1score += win\_points

else:

p2score += win\_points

# tyrell hue

if tyrell\_sum == 3:

if Tyrell1 == 2:

p1score += win\_points

else:

p2score += win\_points

else:

p1score += max(0, diif\_mull \* (f(Tyrell1) \* 2) - f(Tyrell2) \* 2)

p2score += max(0, diif\_mull \* (f(Tyrell2) \* 2) - f(Tyrell1) \* 2)

# baratheon hue

if baratheon\_sum == 4 or Baratheon1 > 2 or Baratheon2 > 2:

if Baratheon1 > 2 or (Baratheon1 == 2 and *player1*.last["Baratheon"] == 1):

p1score += win\_points

else:

p2score += win\_points

elif baratheon\_sum < 3:

p1score += max(0, diif\_mull \* (f(Baratheon1)) - f(Baratheon2))

p2score += max(0, diif\_mull \* (f(Baratheon2)) - f(Baratheon1))

# targaryen hue

if targaryen\_sum == 5 or Targaryen1 > 2 or Targaryen2 > 2:

if Targaryen1 > 2:

p1score += win\_points

else:

p2score += win\_points

else:

p1score += max(0, diif\_mull \* (f(Targaryen1) / 3) - f(Targaryen2) / 3)

p2score += max(0, diif\_mull \* (f(Targaryen2) / 3) - f(Targaryen1) / 3)

# lannister hue

if lannister\_sum == 6 or Lannister1 > 3 or Lannister2 > 3:

if Lannister1 > 3 or (Lannister1 == 3 and *player1*.last["Lannister"] == 1):

p1score += win\_points

else:

p2score += win\_points

elif lannister\_sum < 5:

p1score += max(0, diif\_mull \* (f(Lannister1) / 4) - f(Lannister2) / 4)

p2score += max(0, diif\_mull \* (f(Lannister2) / 4) - Lannister1 / 4)

# greyjoy hue

if greyjoy\_sum == 7 or Greyjoy1 > 3 or Greyjoy2 > 3:

if Greyjoy1 > 3:

p1score += win\_points

else:

p2score += win\_points

else:

p1score += max(0, diif\_mull \* (f(Greyjoy1) / 4) - f(Greyjoy2) / 4)

p2score += max(0, diif\_mull \* (f(Greyjoy2) / 4) - f(Greyjoy1) / 4)

# stark hue

if stark\_sum == 8 or Stark1 > 4 or Stark2 > 4:

if Stark1 > 4 or (Stark1 == 4 and *player1*.last["Stark"] == 1):

p1score += win\_points \* 1.001

else:

p2score += win\_points \* 1.001

elif stark\_sum < 7:

p1score += max(0, diif\_mull \* (f(Stark1) / 4) - f(Stark2) / 4)

p2score += max(0, diif\_mull \* (f(Stark2) / 4) - f(Stark1) / 4)

# if p1score > p2score:

# print(p1score - p2score)

return p1score - p2score

اگر ended روشن بود که به صورت واضح برنده را حساب می‌کنیم (یک تغییر هم در player ایجاد کرده ایم که متغیر لست است که می‌گوید ایا اخرین کار برداشته شده از ان خانه را ان بازیکن برداشته یا نه)

اگر هم خاموش بود می‌اییم موارد زیر را چک می‌کنیم.

اگر از آن لحظه قطعا خانه‌ای را بازی کنی برده باشد امتیاز win\_point را می‌گیرد (با ازمون و خطا به ۱۰ رسیدیم)

اگر نبرده باشد. برای تعداد کارت‌هایی که هر کس برداشته باشد. می‌دانیم با برداشتن ‌بیشتر از نصف برنده می‌شود. پس مثلا اگر برای برد نیاز به n کارت باشد و ما m کارت داشته باشیم اول تصمیم گرفتیم به ان امتیاز m/n ام دهیم. اما بعدی مشاهده کردیم که به جای سرمایه گذاری در یک خانه کارت ‌های خود را پخش می‌کرد و تعداد زیادی را نزدیک می‌باخت. به همین دلیل گفتیم که امتیاز کارت اول هر خانه 1/n, دومی 2/n و .. باشد. یعنی اگر m تا برداشته باشیم (m \* (m + 1) / 2) بر n خواهد بود. بعد از اعمال این ایده دیدیم که کمی ارجهییت به خانه‌های با کارت‌های بیشتر داده شده به همین دلیل کمی ضریب خانه‌های با کارت کمتر را بیشتر کردیم. دو نکته باقی می‌ماند. اول edge کیسی که در حالت n زوج مثلا ۸ یکی نصف یعنی ۴ و دیگری ۳ داشته باشد. این حالت را کلا دری هیوریستیک حساب نمی‌کنیم. چون درست است که اگر حالا تمام بشود بازی نفر چهارتایی می‌برد ولی اگر کارت دیگری برداشته شود هر کس اخری را بر دارد می‌برد. با تست کردن هم دیدیم که حساب نکردن این شرط بهتر است.

ایده اخر این است که در یک خانه اگر رقیب ما نیز تعدادی کارت برداشته باشد شاید بهتر است در ان وارد رقابت نشویم. به همین دلیل ضریب تجربی diif\_mul را ان امتیاز حریف را کم کنیم. ان را با صفر ماکس می‌گیریم که عدد منفی نشود.

این ایجنت در اکثر موارد می‌برد. در تست‌های من در ۱۲ تا بازی ۱۰ بار برد.