به نام خدا

گزارش پروژه دوم هوش مصنوعی

علی مهرانی

شماره دانشجویی : 810198542

- بخش Game

- تابع minimax

در این بخش ابتدا باید الگوریتم minimax را پیاده سازی کنیم که برای این کار تابعی به نام minimax در کلاس Othello پیاده سازی میکنیم.
تابع minimax طراحی شده به عنوان ورودی عمق یا depth را دریافت میکند و پس از تشخیص نوبت فعلی ، min یا max را بررسی و تابع minmax را بار دیگر با یک عمق کم تر و نوبت حریف به ازای تمام حرکات ممکن اجرا میکند.
الگوریتم minimax برای مقایسه state های تولید شده از heuristic هر state استفاده میکند که برای تولید این heuristic نیاز به یک تابع تولید کننده آن داریم.

- تابع get_heuristic

تابع get_heuristc را طراحی میکنیم که در هر state مقدار heuristic آن را برمیگرداند.
این که چه مواردی را برای محاسبه heuristic در این تابع در نظر گرفته ایم را در بخش سوالات پاسخ می دهیم.
تابع minimax در نهایت بهترین حرکت min یا max به همراه مقدار heuristic آن را بر میگرداند.

```
def minimax(self, depth) :

    if(depth==0) :
        returnedVal = (self.get_heuristic(), None)
        return returnedVal

    final_move_human, final_move_cpu = None, None

    if(self.current_turn == 1) :
        max_res = -sys.maxsize
        for move in self.get_valid_moves(1) :
            new_board = copy.deepcopy(self.board)
            new_turn = copy.deepcopy(self.current_turn)

            new_state = Othello(False, depth)
            new_state.board = new_board
            new_state.current_turn = new_turn

            new_state.make_move(self.current_turn,move)
            new_state.current_turn = -self.current_turn
            max_temp = new_state.minimax(depth-1)[0]
            if(max_temp >= max_res) :
                max_res, final_move_human = max_temp, move
        return max_res, final_move_human

    else :
        min_res = sys.maxsize
        for move in self.get_valid_moves(-1) :
            new_board = copy.deepcopy(self.board)
            new_turn = copy.deepcopy(self.current_turn)

            new_state = Othello(False, depth)
            new_state.board = new_board
            new_state.current_turn = new_turn

            new_state.make_move(self.current_turn, move)
            new_state.current_turn = -self.current_turn
            min_temp = new_state.minimax(depth-1)[0]
            if(min_temp <= min_res) :
                min_res, final_move_cpu = min_temp, move
        return min_res, final_move_cpu
```

تابع minmax

```python
def get_heuristic(self) :
    ai_tiles, human_tiles = 0, 0
    ai_corners, human_corners = 0, 0
    ai_loose_rows_num, human_loose_rows_num = 0, 0
    num_of_human_possible_moves, num_of_ai_possible_moves = 0, 0
    heuristic = 0

    for i in range(0, 5) :
        for j in range (0, 5) :
            if(self.board[i][j] == -1) :
                ai_tiles += 1
            elif(self.board[i][j] == 1):
                human_tiles += 1

    if(human_tiles + ai_tiles == 36) :
        if(human_tiles > ai_tiles) :
            return sys.maxsize * (1)
        elif(human_tiles < ai_tiles) :
            return -sys.maxsize * (1)
        else :
            return 0 # match draw case

    heuristic += (human_tiles - ai_tiles)*1

    for i in range(0, 10, 5) :
        for j in range(0, 10, 5) :
            if(self.board[i][j] == 1) :
                human_corners += 1
            elif(self.board[i][j] == -1) :
                ai_corners += 1

    heuristic += 5*(human_corners - ai_corners) * 1 #coeff can be more than 5 too !

    num_of_human_possible_moves = len(self.get_valid_moves(1))
    num_of_ai_possible_moves = len(self.get_valid_moves(-1))

    heuristic += 3*(num_of_human_possible_moves - num_of_ai_possible_moves)

    return heuristic
```

get_heuristic تابع

- تابع pruning minimax

این تابع به طور کلی مشابه minimax عمل میکند اما آن state هایی که به دلیل مقدار heuristic شان و تفاوتشان با مقدار alpha و beta نیاز به بررسی شدن ندارند ، کنار گذاشته میشوند و بررسی نمیشوند.
تابع pruning minimax مطابق شکل زیر عمل میکند.

```python
def pruning_minimax(self, depth, max_val, min_val) : #alpha -> max_val , beta -> min_val
    if(depth==0) :
        returnedVal = (self.get_heuristic(), None)
        return returnedVal

    final_move_human, final_move_cpu = None, None

    if(self.current_turn == 1) :
        max_res = -sys.maxsize
        for move in self.get_valid_moves(1) :
            new_board = copy.deepcopy(self.board)
            new_turn = copy.deepcopy(self.current_turn)
            new_state = Othello(False, depth)
            new_state.board = new_board
            new_state.current_turn = new_turn
            new_state.make_move(self.current_turn,move)
            new_state.current_turn = -self.current_turn

            max_temp = new_state.pruning_minimax(depth-1, max_val, min_val)[0]
            if(max_temp >= max_res) :
                max_res, final_move_human = max_temp, move
            if(max_res >= min_val) :
                break
            max_val = max(max_val, max_res)

        return max_res, final_move_human

    else :
        min_res = sys.maxsize
        for move in self.get_valid_moves(-1) :
            new_board = copy.deepcopy(self.board)
            new_turn = copy.deepcopy(self.current_turn)
            new_state = Othello(False, depth)
            new_state.board = new_board
            new_state.current_turn = new_turn
            new_state.make_move(self.current_turn, move)
            new_state.current_turn = -self.current_turn

            min_temp = new_state.pruning_minimax(depth-1, max_val, min_val)[0]
            if(min_temp <= min_res) :
                min_res, final_move_cpu = min_temp, move
            if(min_res <= max_val) :
                break

            min_val = min(min_val, min_res)
        return min_res, final_move_cpu
```

تابع pruning minimax

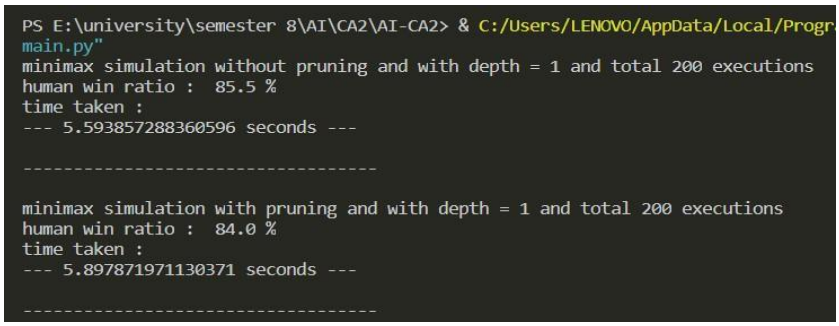
- تابع get_human_move

در تابع get_human_move در صورتی که مقدار شاخص prune برابر true بود از تابع pruning_minimax و در صورتی که false بود از تابع minimax استفاده میکنیم.

- نتایج شبیه سازی

- عمق 1

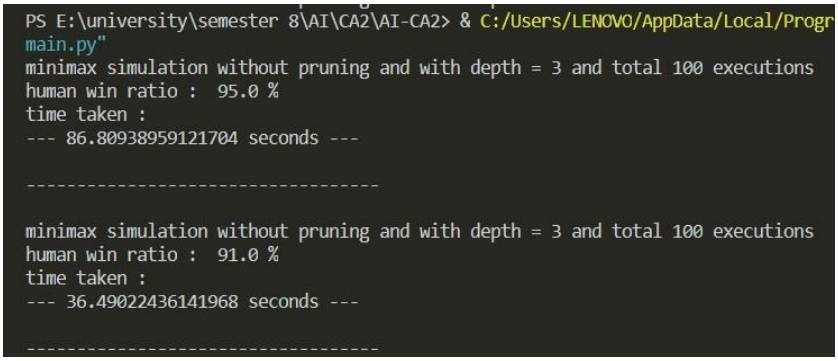
ابتدا برای عمق 1 بازی را هم با pruning و هم بدون pruning برای 200 بار اجرا میکنیم که نتایج آن به صورت زیر است.



همانگونه که مشخص است بدون هرس کردن شانس برد 85 درصد و با هرس کردن شانس آن 84 درصد است.

- عمق 3

برای عمق 3 نیز نتایج پس از 100 با اجرا به صورت زیر است



همانگونه که مشخص است بدون هرس کردن شانس برد 95 درصد و با هرس کردن شانس آن 91 درصد است.

- عمق 5

بدلیل طولانی بودن زمان اجرا الگوریتم در عمق 5 بدون هرس کردن، بازی را 10 بار اجرا میکنیم که نتایج آن به صورت زیر است

```
PS E:\university\semester 8\AI\CA2\AI-CA2> & C:/Users/LENOVO/AppData/Local/Prog
main.py"
minimax simulation without pruning and with depth = 5 and total 10 executions
human win ratio :  100.0 %
time taken :
--- 361.6030008792877 seconds ---

-----------------------------------

minimax simulation with pruning and with depth = 5 and total 10 executions
human win ratio :  90.0 %
time taken :
--- 50.24600839614868 seconds ---

-----------------------------------
```

همانگونه که مشخص است بدون هرس کردن شانس برد 100 درصد و با هرس کردن شانس آن 90 درصد است.

- عمق 7

به دلیل زمان زیاد برای اجرا در این عمق این الگوریتم را 10 بار اجرا میکنیم که یک بار 100 و بار دیگر 90 درصد است

```
PS E:\university\semester 8\AI\CA2> & C:/Users/LE
"
human win ratio : (alpha-beta with depth=7) 1.0
time taken for :
--- 538.8248224258423 seconds ---
PS E:\university\semester 8\AI\CA2>
```

```
PS E:\university\semester 8\AI\CA2\AI-CA2> & C:/Users/LENOVO/AppData/Local/P
main.py"
minimax simulation with pruning and with depth = 7 and total 10 executions
human win ratio :  90.0 %
time taken :
--- 525.8564977645874 seconds ---
```

- توجه

لازم به ذکر است که نتایج بازی و شانس برد در هر دور اجرا ممکن است متفاوت باشد و وابسته به تعداد اجراها نیز می باشد. نرخ پیروزی در هر عمق نیز ممکن است نسبت به شبیه سازی قبلی بیش تر و یا کم تر باشد مثلا برای عمق 7 می تواند یک بار 100 باشد و در شبیه سازی بعدی 90 اما به طور کلی می توان گفت که با افزایش عمق شانس پیروزی افزایش می یابد.

- پاسخ سوالات

1. برای محاسبه heuristic به طور کلی سه عامل را در نظر گرفته شده و به هر کدام وزنی داده شده که این سه عامل موارد زیر هستند.

- اختلاف تعداد مهره های خود با حریف با ضریب 1
(میتوانستیم اختلاف را نیز در نظر نگیریم و فقط تعداد مهره های گرفته شده را توسط بازیکن 1 را محاسبه کنیم)

- اختلاف تعداد مهره های خود در گوشه زمین با حریف ، با ضریب 5
(در اینجا نیز می توانستیم اختلاف را در نظر نگیریم و فقط تعداد مهره های بازیکن 1 در گوشه زمین را محاسبه کنیم)
گرفتن مهره در گوشه زمین بازی از اهمیت بالایی برخوردار است

- تعداد حرکات ممکن در state مورد نظر پس از انجام حرکت و اختلاف آن با تعداد حرکات ممکن حریف ، با ضریب 3
بهتر است حرکتی انجام شود که پس از آن تعداد حرکات های ممکن حریف کم باشد و تعداد حرکات ما نیز پس از حرکت حریف نیز کم نباشد.

مقدار heuristic نهایی جمع وزن دار این سه مورد است

- در صورت پر شدن تمام صفحه توسط مهره ها در صورتی که تعداد مهره های بازیکن از cpu بیش تر باشد بالاترین مقدار ممکن بر گردانده میشود که برای برد بازیکن است و در حالت برعکس یعنی وقتی cpu برنده میشود این مقدار کم ترین عدد ممکن است، در صورت تساوی نیز 0 برگردانده میشود.

2. بله ، عمق الگوریتم نیز موثر است. با افزایش عمق شانس برد بیش تر میشود اما از طرفی تعداد state های دیده شده و همچنین زمان مصرفی نیز بسیار بیش تر میشود

3. ترتیب فرزندان در میزان هرس موثر است و در حالتی که بهترین حرکت در هر بخش در سمت چپ (چپ ترین) درخت باشد بیش تربن میزان هرس رخ می دهد که برای رسیدن به این حالت باید ترتیب دیدن هر state در هر بخش بر اساس مقدار heuristic ان باشد در واقع بهترین node ها باید در ابتدا چک شوند.

4. branching factor در واقع تعداد فرزندان در هر مرحله از درخت است که در این مسئله به طور کلی به معنای تعداد حرکات ممکن در هر مرحله یا state بازی است که در ابتدا تعداد حرکات ممکن کم است و با گذشت بازی افزایش می یابد اما از طرفی با نزدیک شدن به پایان بازی نیز تعداد حرکات ممکن و در نتیجه branching factor کاهش می یابد.

5. عملیات هرس کردن بر روی زیر درخت ها و برگ هایی رخ میدهد که نیاز به بررسی شدن ندارند و بهترین node در زیر درخت پیدا شده و بررسی آن ها دیگر نیاز نیست و در این حالت میزان دقت به طور کلی کاهش نمی یابد چون که فقط node هایی هرس میشوند که نیاز به بررسی ندارند و جواب optimal ارائه نمیدهند ، عملیات هرس نیز سرعت را به طور کلی افزایش میدهد.

6. به این خاطر که در minimax فرض میشود که حریف نیز تلاش در انتخاب کم ترین heuristic برای ما را دارد که در این حالت حریف نیز گویا از minimax استفاده میکند و بهترین حرکت ممکن برای خودش را انتخاب میکند اما در این مسئله

که حریف به صورت random حرکت میکند احتمال انتخاب بهترین حرکت ممکن توسط آن زیاد نیست و با فرض optimal بودن آن در واقع الگوریتم minimax برای حرکت او اضافی اجرا میشود که باعث کاهش سرعت میشود
در واقع minimax که با فرض optimal بودن هر دو اجرا میشود برای حریف غیر optimal بیهوده اجرا میشود
برای بهتر کردن الگوریتم در این حالت میتوان یک حد میانگین از حرکت حریف بدست اورد یعنی حرکتی که در وسط حرکات ممکن مرتب شده با heuristic باشند و بر اساس آن حرکت و در واقع میانگین heuristic حرکات مختلف، حرکت خود را انجام داد و در این حالت تمام حرکات انجام شده توسط ان را نیاز نیست بررسی کنیم که سرعت الگوریتم را افزایش میدهد.