مجید فریدفر

۸۱۰۱۹۹۵۶۹

گزارشکار تمرین کامپیوتری دوم هوش مصنوعی (ترم بهار ۱۴۰۲)

بخش دوم: بازی

در ابتدا دو تا متودهای کلاس Othello را به صورت تابع در بیرون از بدنه‌ی کلاس تعریف می‌کنیم:

- Make_move: به جای این که حرکت را روی self.board اعمال بکند، یک board به عنوان ورودی میگیرد. سپس یک کپی از روی آن میگیرد به اسم new_noard و تغییرات را روی آن اعمال کرده و این board جدید را برمیگرداند. در واقع این کار را به صورت محازی انجام میدهد.
- Get_valid_moves: به جای این که حرکت های ممکن روی self.board را پیدا بکند، یک board ورودی میگیرد و حرکت هایی ممکن روی آن را برمیگرداند.

همچنین یک متود جدید در این کلاس اضافه میکنیم به اسم value_. این متود در واقع مقدار minimax یک نود را روی درخت برمیگرداند. ذکر چند نکته در اینجا الزامی است:

- در این مسئله، nodeهای درخت در واقع boardهای مختلفی است که در طی بازی به آنها میرسیم. **پس یک ورودی این متود، board است.**
- چون نمیخواهیم تا آخرین عمق درخت را بررسی کنیم، از یک متغیر minimax_depth استفاده میکنیم. به این شکل که اگر به این عمق در درخت رسیدیم، دیگر پایین تر نمیرویم. و به جای محاسبه‌ی value_، به یک مقدار heuristic اکتفا میکنیم که صرفا یک حدس برای value_ است. (این کار را زمانی که به یکی از terminalها رسیدیم هم انجام میدهیم).
- این متود سه تا مقدار خروجی دارد:
  - مقدار valueی minimax راس
  - بهترین حرکت در این state (که با توجه به value تعیین میشود).
  - تعداد node (board)های دیده شده از این راس

```
    def _value(self, board, player, depth, a = None, b = None):
        if depth == 0:
            return heuristic(board, self.size), None, 1

        if len(get_valid_moves(board, self.size, 1)) == 0 and
len(get_valid_moves(board, self.size, -1)) == 0:
            return heuristic(board, self.size), None, 1

        best_move = None
        visiteds = 1
```

```
        v = 0

        if player == 1:
            v = -math.inf
        else:
            v = +math.inf

        for m in get_valid_moves(board, self.size, player):
            new_board = make_move(board, self.size, player, m)
            value, _m, _v = self._value(deepcopy(new_board), 1 if player == -1 else -
1, depth-1, a, b)

            if player == 1:
                if value > v:
                    v = value
                    best_move = m

                if self.prune:
                    if value >= b:
                        v = value
                        best_move = m
                        break

                    a = max(a, value)

            else:
                if value < v:
                    v = value
                    best_move = m

                if self.prune:
                    if value <= a:
                        v = value
                        best_move = m
                        break

                    b = min(b, value)

            visiteds += _v

        return v, best_move, visiteds
```

این متود پیاده سازی روش minimax است. دقت میکنیم:

- تابع get_valid_moves در واقع مسیر رسیدن به فرزندان state کنونی را میدهد. با استفاده از تابع make_move میتوانیم این فرزندان را تولید کنیم و از valueشان استفاده کنیم بدون دست زدن به self.board.
- Player اول، میخواهد value را بیشینه کند.
- Player دوم، میخواهد کمینه کند.

نتایج (کد در هر حالت، ۱۰۰ بار ران شده است. پس عدد total wins درصد شانس پیروزی است).

- بدون هرس – عمق ۱:

```
##################################
Total Wins: 63
Average Time: 0.01644693613052368
Average Visited Nodes: 95.63
```

- بدون هرس – عمق ۳:

```
##################################
Total Wins: 74
Average Time: 0.49236836910247805
Average Visited Nodes: 3187.83
```

- بدون هرس – عمق ۵:

```
##################################
Total Wins: 96
Average Time: 13.468159372806548
Average Visited Nodes: 120326.5
```

- با هرس – عمق ۱:

```
##################################
Total Wins: 67
Average Time: 0.02110533952713013
Average Visited Nodes: 94.93
```

- با هرس – عمق ۳:

```
##################################
Total Wins: 81
Average Time: 0.29717994928359986
Average Visited Nodes: 808.86
```

- با هرس – عمق ۵:

```
##################################
Total Wins: 96
Average Time: 1.4724567532539368
Average Visited Nodes: 4717.11
```

- با هرس – عمق ۷:

```
##################################
Total Wins: 98
Average Time: 21.05507731437683
Average Visited Nodes: 24884.22
```

همچنین فایل کامل اجراها به صورت ipynb. در codes/tests هست.

سوال‌ها:

۱. محاسبه‌ی آن ساده و سریع باشد.
تخمین خوبی از امتیاز بدهد.

۲. بله. هرچه عمق الگوریتم بیشتر باشد، یعنی ایجنت ما آینده نگری بیشتری دارد و میتواند بهتر تصمیم بگیرد و تخمین بهتری بزند. برای همین شانس پیروزی بیشتر میشود. اما چون بررسی این حالت ها زمان بر است و باید نودهای بیشتری ببیند و بررسی کند، زمان اجرای الگوریتم زیاد میشود.

۳. بله میتوانیم. دو روش برای انجام این کار داریم. که حافظه ی مصرفی برنامه را افزایش میدهند ولی زمان اجرای الگوریتم (به علت افزایش تعداد هرس‌ها) کاهش میابد.
روش اول - مرتب سازی ایستا:
فرزندان هر نود را بر اساس مقدار فیتنس و هیوریستیکشان مرتب میکنیم. سپس در هر مرحله بهینه ترین را میبینیم و ادامه میدهیم.
روش دوم – مرتب سازی پویا:
وقتی داریم سرچ میکنیم، ترتیب گره ها را به ترتیبی تغییر میدهیم که در درست ترین حالت قرار بگیرند. به این ترتیب میتوانیم بیشترین هرس را داشته باشیم. این اتفاق برحسب وضعیت کنونی بازی محاسبه و اعمال میشود. مثلا اگر در حین سرچ کردن، نودی پیدا کنیم که هیوریسیتک بالایی دارند، اولویت دیده شدن آن بالاتر میرود.

۴. به حداکثر تعداد حرکت های ممکن در هر مرحله (در کل بازی) برنچینگ فکتور میگویند. یا به بیان دیگر حداکثر تعداد فرزندان یک نود در طول سرچ کردن.
در ابتدا این مقدار، زیاد است. چون مسیرهای زیادی برای حرکت داریم. اما به مرور و با حرکت کردن، صفحه پر میشود و تعداد مسیرها کاهش میابد. همچنین با اعمال الگوریتم مینیمکس و هرس کردن تعدادی از مسیرها حذف خواهند شد. با این ترتیب در طول برنامه، برنچینگ فکتور کاهش میابد.

۵. چون با هرس کردن، بخش هایی از درخت را حذف میکنیم که مطمئنیم تاثیری در نیتجه نهایی محاسباتمان ندارد. به این ترتیب بدون از دست دادن دقت، سرعت بالا میرود.

۶. چون در این الگوریتم فرض بر این است که حریف بهترین حرکت خودش را انجام میدهد. اما چن حریف این طور نیست، ما شانس این را داشتیم که با انجام بازی شکلی دیگر امتیاز بیشتری بگیریم. (مثال در اسلایدها هست).

میتوانیم به جای ماکسیمم گرفتن بین ولیوی فرزندان گره فعلی، میانگینشان را حساب کنیم. به این ترتیب چون حرکت حریف تصادفی است، شانس این را داریم که به نتیجه ای بهتر برسیم.
همچنین میتوانیم از روش Monte Carlo Tree Search استفاده کنیم.