گزارش پروژه بازی

آوا میرمحمدمهدی 810199501

توضیح نحوه‌ی نوشتن تابع minimax:

در این تابع درصورتی که عمق 0 بود یا دیگر حرکتی برای انجام باقی نمانده بود، مقدار evaluate را بازمی‌گردانیم؛ سپس برای ادامه‌ی بازی دو حالت وجود دارد: حساب کردن مینیمم امتیاز برای وقتی که حریف (آبی) بازی می‌کند و محاسبه ماکسیمم امتیاز برای وقتی که agent ما (قرمز) بازی می‌کند. در هر بار اجرای minimax باید به ازای هر حرکتی که در صفحه باقی مانده است چک کنیم که با انجام آن چه امتیازی بدست خواهیم آورد؛ برای اینکار در ابتدا آن را از available_moves حذف می‌کنیم و آن حرکت را در red یا blue اضافه می‌کنیم، تابع minmax را با عمق 1 واحد کمتر از عمق فعلی فراخوانی می‌کنیم و پس ار فراخوانی تابع، حرکتی که از available_moves حذف کرده بودیم را به آن برمی‌گردانیم تا وقتی بعدا مجددا به این مرحله backtrack می‌کنیم همچنان در لیست available_moves موجود باشد. (برای لیست red یا blue و نوبت دهی به آبی یا قرمز نیز همین کار را می‌کنیم). درصورتی که نوبت قرمز بود، اگر امتیاز کسب شده بیشتر از حداکثر امتیاز کسب شده تا کنون بود، آن را جایگزین می‌کنیم و درصورتی که نوبت آبی بود برعکس این‌کار را انجام می‌دهیم.

```python
def minimax(self, depth, player_turn, alpha, beta):
        if depth == 0 or len(self.available_moves) == 0:
            return None, self._evaluate(), depth
        best_move = None
        best_move_depth = math.inf
        if player_turn == 'red':
            best_score = -math.inf
        else:
            best_score = math.inf
        for move in deepcopy(self.available_moves):
            self.available_moves.remove(move)
            if player_turn == 'red':
                self.red.append(move)
            else:
                self.blue.append(move)
            self.turn = self._swap_turn(self.turn)
            _, score , depth_= self.minimax(depth - 1, self.turn, alpha, beta)
            self.available_moves.append(move)
            if player_turn == 'red':
                self.red.remove(move)
            else:
                self.blue.remove(move)
            self.turn = self._swap_turn(self.turn)
            if player_turn == 'red':
                if score > best_score or (score == best_score and depth_ < best_move_depth):
                    best_move_depth = depth_
                    best_score = score
                    best_move = move
                alpha = max(alpha, best_score)
            else:
                if score < best_score or (score == best_score and depth_ < best_move_depth):
                    best_move_depth = depth_
                    best_score = score
                    best_move = move
                beta = min(beta, best_score)
            if self.prune and beta <= alpha:
                break
        return best_move, best_score, best_move_depth
```

توضیح چگونگی پیاده‌سازی هرس آلفا و بتا:

در ابتدا و در تابع play، مقدار آلفا را برابر با منفی بی نهایت و مقدار بتا را برابر با مثبت بی نهایت قرار می‌دهیم و آنها را به عنوان پارامتر به تابع minimax پاس می‌دهیم. در تابع minimax درصورتی که نوبت قرمز باشد، مقدار آلفا را برابر با ماکسیمم مقدار آلفا و بهترین امتیاز کسب شده تا آن زمان قرار می‌دهیم و درصورتی که نوبت آبی باشد، مقدار بتا را برابر با مینیمم مقدار بتا و بهترین امتیاز تا آن زمان قرار می‌دهیم و درصورتی که مقدار بتا کوچکتر یا مساوی آلفا بود و prune برابر با True بود، هرس را انجام می‌دهیم و دیگر بقیه‌ی مقادیر آن زیرشاخه را بررسی نمی‌کنیم.

سوالات:

1) یک heuristic خوب چه ویژگی‌هایی دارد؟ علت انتخاب heuristic شما و دلیل برتری آن نسبت به تعدادی از روش‌های دیگر را بیان کنید.

یک هیوریستیک خوب در این مساله احتمال برد را بالا می‌برد و باید در این مساله با باخت و برد ارتباط داشته باشد. کد هیوریستک انتخاب شده در تصویر زیر نشان داده شده است. در این هیوریستیک درصورتی که agent ما بازی را باخته باشد مقدار منفی بی‌نهایت و درصورتی که بازی را برده باشد مقدار مثبت بی‌نهایت برگردانده می‌شود؛ اگر دو حالت گفته شده اتفاق نیفتاده بود، حرکاتی که agent می‌تواند انجام دهد را بررسی می‌کنیم، درصورتی که این حرکت باعث برد و باخت نشد 4 امتیاز مثبت و درصورتی که باعث باخت agent ما شد 2 امتیاز منفی در نظر می‌گیریم (برای رنگ آبی دقیقا برعکس اینکار را انجام می‌دهیم و جمع و یا منها بودن نیز برعکس است.) هیوریستیک دیگر میتواند مجموع درجه رئوس بزرگتر از 1 به رنگ قرمز و منفی مجموع درجه رئوس بزرگتر از 1به رنگ آبی باشد ولی در این حالت وقتی سه یال داشته باشیم به طوری که دو به دو در یک راس مشترک باشند مقدار هیوریستیک بیشتر از زمانی است که سه یال در یک راس مشترک باشند.

```
def _evaluate(self):
        gameoverValue = self.gameover(self.red, self.blue)
        if gameoverValue == 'red':
            return math.inf
        if gameoverValue == 'blue':
            return -math.inf
        movesWithoutTriangle = 0
        for move in deepcopy(self.available_moves):
            if self.turn == 'red':
                self.red.append(move)
                self.available_moves.remove(move)
                gameoverValue = self.gameover(self.red, self.blue)
                if gameoverValue == 0:
                    movesWithoutTriangle += 4
                if gameoverValue == 'blue':
                    movesWithoutTriangle -= 2
                self.red.remove(move)
                self.available_moves.append(move)
            else:
                self.blue.append(move)
                self.available_moves.remove(move)
                gameoverValue = self.gameover(self.red, self.blue)
                if gameoverValue == 0:
                    movesWithoutTriangle -= 4
                if gameoverValue == 'red':
                    movesWithoutTriangle += 2
                self.blue.remove(move)
                self.available_moves.append(move)
        return movesWithoutTriangle
```

2) آیا میان عمق الگوریتم و پارامترهای حساب شده روابطی می‌بینید؟ به طور کامل بررسی کنید که عمق الگوریتم چه تاثیراتی بر روی شانس پیروزی، زمان و گره‌های دیده شده می‌گذارد.

در الگوریتم minimax، ما فرض می‌کنیم که حریف بهینه بازی می‌کند درصورتی که اینطور نیست؛ پس وقتی برنامه را با عمق‌های بالاتر اجرا می‌کنیم، چون این فرض می‌شود که حریف بهینه است و مراحل تا عمق‌های بالاتر را می‌بینیم، درصورتی که در عمق‌های بیشتر، agent باخت خودش را ببیند، مقدار منفی بی‌نهایت برگردانده می‌شود و دیگر برایش فرقی ندارد که این مقدار در چه عمقی برگردانده شده است؛ در این حالت agent در همان ابتدا که این موضوع را می‌فهمد باعث باخت خودش می‌شود درصورتی که اگر اینکار را نکرده بود، ممکن بود حریف حرکت بهینه را انجام ندهد و agent ما بازی را ببرد؛ با توضیحات گفته شده وقتی عمق را بیشتر می‌کنیم احتمال برد agent ما کاهش می‌یابد. برای حل کردن نسبی این مشکل، می‌توان عمق را نیز در هر مرحله از تابع بازگشتی return کرد و درصورتی که در امتیاز حرکتی برابر با بهترین امتیاز بود تنها درصورتی حرکت را تغییر دهیم که در عمق بیشتری از بهترین امتیاز قبلی قرار داشته باشیم؛ اینکار باعث می‌شود تا از باخت زودهنگام agent جلوگیری شود. بدیهی است که هرچه عمق پیمایش شده بیشتر باشد،گره‌های دیده شده نیز بیشتر خواهد بود و زمان اجرای برنامه افزایش می‌یابد؛ این افزایش زمان مخصوصا برای اولین حرکت انجام شده در بازی بسیار قابل توجه خواهد بود زیرا تعداد حرکت‌های قابل انجام در این مرحله بیشترین مقدار است.

3) وقتی از روش هرس کردن استفاده می‌کنید، برای هر گره درخت، فرزندانش به چه ترتیبی اضافه می‌شوند؟ آیا این ترتیب اهمیت دارد؟ چرا این ترتیب را انتخاب کردید؟

در استفاده از هرس در بعضی از موارد، ترتیب اهمیت دارد؛ درواقع ممکن است بعضی مواقع چینش درخت اولیه به گونه‌ای باشد که هرس کردن و نکردن تفاوت زیادی نداشته باشند و گره‌هایی که قابل حذف شدن باشند در انتها قرار گرفته باشند (و در واقع در ابتدا حذف نشوند و زمان کاهش نیابد.) در این مساله یال‌های ما که همان حرکت‌های قابل انجام هستند، به ترتیب موقعیت رئوس مرتب شده‌اند و در هربار به همین ترتیب پیمایش می‌شوند و احتمال مقدار امتیاز کسب شده در همه‌ی آنها در ابتدا یکسان است و چون رئوس به ترتیب قرار گرفته اند، احتمال نزدیک شدن چینش به تشکیل مثلث وجود دارد و درنتیجه هرس آلفا و بتا کمک کننده خواهد بود.

بدون استفاده از هرس آلفا و بتا:

عمق 1:

```
PS D:\Term5\AI\CA2> python main.py 1 0
game played for 100 times
{'red': 91, 'blue': 9}
Algorithm time: 0.003058505058288574
winning probability: 0.91
```

عمق 3:

```
PS D:\Term5\AI\CA2> python main.py 3 0
game played for 100 times
{'red': 74, 'blue': 26}
Algorithm time: 0.26162468910217285
winning probability: 0.74
```

عمق 5:

```
PS D:\Term5\AI\CA2> python main.py 5 0
game played for 100 times
{'red': 63, 'blue': 37}
Algorithm time: 24.66349313259125
winning probability: 0.63
```

با استفاده از هرس آلفا و بتا:

عمق 1:

```
PS D:\Term5\AI\CA2> python main.py 1 0
Algorithm time: 0.0564875419873047
winning probability: 0.88
```

عمق 3:

```
PS D:\Term5\AI\CA2> python main.py 3 0
game played for 100 times
{'red': 85, 'blue': 15}
Algorithm time: 0.058362743854522704
winning probability: 0.85
```

عمق 5:

```
PS D:\Term5\AI\CA2> python main.py 5 0
game played for 100 times
{'red': 70, 'blue': 30}
Algorithm time: 0.6956777834892273
winning probability: 0.7
```

عمق 7:

```
PS D:\Term5\AI\CA2> python main.py 7 0
game played for 100 times
{'red': 65, 'blue': 35}
Algorithm time: 8.174740545749664
winning probability: 0.65
```

همانطور که میبینیم درصد شانس پیروزی در زمان استفاده از هرس آلفا و بتا و در زمان عدم استفاده از هرس آلفا و بتا تقریبا یکسان است. توضیحات راجع به زمان و مقدار درصد شانس در بالاتر داده شده است.

یکی از تغییرات اعمال شده در کد اصلی، تغییر در تابع gameover است زیرا در نسخه قبلی آن مشکلی وجود داشت و اگر اندازه لیست r کوچکتر از 3 بود 0 ریترن می‌کرد درصورتی که ممکن بود اندازه لیست b 3 باشد و تشکیل مثلث داده باشد. تصویر تابع جدید در زیر آورده شده است. در این تابع یک ست از تمامی نقاط تشکیل شده است و درصورتی که تنها سه عضو غیرتکراری در آن وجود داشته باشد به این معناست که مثلث تشکیل شده است.

```
r.sort()
for i in range(len(r) - 2):
    for j in range(i + 1, len(r) - 1):
        for k in range(j + 1, len(r)):
            dotSet = set()
            dotSet.update(r[i])
            dotSet.update(r[j])
            dotSet.update(r[k])
            if len(dotSet) == 3:
                return 'blue'

b.sort()
for i in range(len(b) - 2):
    for j in range(i + 1, len(b) - 1):
        for k in range(j + 1, len(b)):
            dotSet = set()
            dotSet.update(b[i])
            dotSet.update(b[j])
            dotSet.update(b[k])
            if len(dotSet) == 3:
                return 'red'

return 0
```