سنا ساری نوایی

شماره دانشجویی : 810199435

هوش مصنوعی

پروژه دوم(بازی)

هدف پروژه : آشنایی با الگوریتم min-max از طریق بازی SIM

توضیح کلی: در این پروژه بازی SIM را با استفاده از الگوریتم min-max پیاده سازی میکنیم. این بازی به این صورت است که 6 نقطه داریم و هر یک از بازیکنان آبی و قرمز به نوبت یک راس را به راس دیگر متصل میکنند. اولین بازیکنی که مثلث تشکیل دهد، بازی را میبازد. در این کد، بازیکن قرمز از طریق الگوریتم min-max و بازیکن آبی به صورت رندوم عمل میکند. وقتی یک بازی تمام شود، بازی دیگر بلافاصله بعد از آن شروع میگردد و به همان تعداد که مشخص کردیم، ادامه می یابد و در اخر نتیجه کلی در صفحه نمایان میشود.

تابع play()

در تابع play() روند کلی بازی انجام میشود. در ابتدا مقدار دهی های اولیه انجام میشود. در تابع initialize() صفحه بازی آماده میشود و 6 نقطه ای که قرار است بازی روی آنها انجام شود، رسم میشود و همچنین انتخاب میشود که کدام بازیکن اول بازی را آغاز کند. به تابع play() برمیگردیم. پس از initialize() ، چرخه ی روبرو هر بار طی میشود تا زمانیکه بازی تمام شود. ابتدا بررسی میکنیم نوبت کدام بازیکن است. اگر نوبت بازیکن قرمز بود، از min-max استفاده میکنیم و اگر نوبت بازیکن آبی بود، از مقدار رندوم بهره میبریم. پس از آن نوبت را به بازیکن دیگر میدهیم و شکل بازی اپدیت شده را میکشیم. سپس وارد تابع gameover() میشویم و اگر مثلث تشکیل شده باشد، بازی پایان می‌یابد.

```
def play(self):
    self.initialize()
    while True:
        if self.turn == 'red':
            selection = self.minimax(depth=self.minimax_depth, player_turn=self.turn)[0]
            if selection[1] < selection[0]:
                selection = (selection[1], selection[0])
        else:
            selection = self.enemy()
            if selection[1] < selection[0]:
                selection = (selection[1], selection[0])
        if selection in self.red or selection in self.blue:
            raise Exception("Duplicate Move!!!")
        if self.turn == 'red':
            self.red.append(selection)
        else:
            self.blue.append(selection)

        self.available_moves.remove(selection)
        self.turn = self._swap_turn(self.turn)
        selection = []
        self.draw()
        r = self.gameover(self.red, self.blue)
        if r != 0:
            return r
```

تابع minimax()

در این تابع به اینصورت عمل میکنیم که در ابتدای تابع چک میکنیم مثلث تشکیل شده است یا خیر. این کار برای وقتی مناسب است که میخواهیم به ازای هر عمق پیش بینی کنیم کدام عمل رخ میدهد و چک کردن مثلث باعث میشود تصمیم گیری بهتری داشته باشیم.سپس به ازای هر بازیکن که نوبتش باشد، minmax را اجرا میکنیم. قرمز به دنبال ماکسیمم کردن امتیاز خودش است. سپس به ازای هر بازیکن که نوبتش باشد، حرص الفا بتا را هم در نظر میگیریم. اگر عددی که بدست آوردیم، از مقدار الفا بیشتر بود، همان را ریترن میکنیم و اگر نوبت بازیکن آبی باشد و عددی که بدست می‌آوریم از بتا کوچکتر باشد، همان را ریترن میکنیم.

```
def minimax(self, depth, player_turn, alpha =-INF, beta = INF):
    #check if a triangle is formed
    if len(self.red) >= 3:
        self.red.sort()
        for i in range(len(self.red) - 2):
            for j in range(i + 1, len(self.red) - 1):
                for k in range(j + 1, len(self.red)):
                    if self.red[i][0] == self.red[j][0] and self.red[i][1] == self.red[k][0] and self.red[j][1] == self.red[k][1]:
                        return None,-INF
    if len(self.blue) >= 3:
        self.blue.sort()
        for i in range(len(self.blue) - 2):
            for j in range(i + 1, len(self.blue) - 1):
                for k in range(j + 1, len(self.blue)):
                    if self.blue[i][0] == self.blue[j][0] and self.blue[i][1] == self.blue[k][0] and self.blue[j][1] == self.blue[k][1]:
                        return None,INF

    #check if depth is 0
    if depth == 0:
        return None, self._evaluate()

    selectedMove = None
    #maximizing player
    if player_turn == 'red':
        maxValue = -INF
        for child in self.available_moves:
            temp = self.set_changes(child, depth,alpha,beta,player_turn)

            #pruning
            if temp < maxValue:
                continue
            selectedMove = child
            maxValue = temp
            alpha = alpha if alpha > maxValue else maxValue
            if self.checkMaxValue(maxValue, beta): return selectedMove, maxValue

        return selectedMove, maxValue
    #minimizing player
    if player_turn == 'blue':
        minValue = INF
        for child in self.available_moves:
            temp = self.set_changes(child, depth,alpha,beta,player_turn)

            #pruning
            if temp > minValue:
                continue
            selectedMove = child
            minValue = temp
            beta = beta if beta < minValue else minValue
            if self.checkMinValue(minValue, alpha): return selectedMove, minValue

        return selectedMove, minValue
```

تابع evaluation()

تابع evaluate() را به صورت زیر پیاده سازی کرده ایم. ابتدا مشخص میکنیم که نوبت کدام بازیکن است. سپس به ازای همه حرکت هایی که انجام دادن آنها امکان پذیر است، چک میکنیم که آیا یال بعدی که اضافه شود، مثلث تشکیل میدهد یا خیر. پس در هر مرحله ابتدا یال را به لیست مربوط به هر رنگ اضافه میکنیم و سپس چک میکنیم مثلث تشکیل میشود یا خیر. اگر نوبت بازیکن قرمز بود و مثلث تشکیل شد، این به نفع ما نیست و امتیاز 3- دریافت میکنیم ولی اگر تشکیل نشد، امتیاز 5+ میگیریم. به همین صورت اگر بازیکن آبی مثلث تشکیل بدهد، به نفع بازیکن قرمز است پس امتیاز 5+ میگیریم و اگر تشکیل ندهد، 3- دریافت میکنیم و در نهایت یال اضافه شده را از لیست در هر مرحله حذف میکنیم و در نهایت امتیاز را ریترن میکنیم.

```
def _evaluate(self):
    score = 0
    if self.turn == 'red':
        for x in self.available_moves:
            self.red.append(x)
            if len(self.red) >= 3:
                self.red.sort()
                for i in range(len(self.red) - 2):
                    for j in range(i + 1, len(self.red) - 1):
                        for k in range(j + 1, len(self.red)):
                            if self.red[i][0] == self.red[j][0] and self.red[i][1] == self.red[k][0] and self.red[j][1] == self.red[k][1]:
                                score -= 3
                            else:
                                score += 5
            self.red.remove(x)
        return score
    if self.turn == 'blue':
        for x in self.available_moves:
            self.blue.append(x)
            if len(self.blue) >= 3:
                self.blue.sort()
                for i in range(len(self.blue) - 2):
                    for j in range(i + 1, len(self.blue) - 1):
                        for k in range(j + 1, len(self.blue)):
                            if self.blue[i][0] == self.blue[j][0] and self.blue[i][1] == self.blue[k][0] and self.blue[j][1] == self.blue[k][1]:
                                score += 5
                            else:
                                score -= 3
            self.blue.remove(x)
        return score
```

حال به ازای عمق برابر 1 سه بار برنامه را اجرا میکنیم. نتایج به صورت زیر میباشد:

```
Time taken: 0.2074441909790039
{'red': 90, 'blue': 10}
```

```
Time taken: 0.27418017387390137
{'red': 86, 'blue': 14}
```

```
Time taken: 0.19661426544189453
{'red': 87, 'blue': 13}
```

به ازای عمق برابر 3 سه بار برنامه را اجرا میکنیم که نتایج بصورت زیر میباشد:

```
Time taken: 2.3712387084960938
{'red': 85, 'blue': 15}
```

```
Time taken: 2.5746753215789795
{'red': 93, 'blue': 7}
```

```
Time taken: 2.4293556213378906
{'red': 92, 'blue': 8}
```

به ازای عمق برابر 5 سه بار برنامه را اجرا میکنیم که نتایج به صورت زیر میباشد:

```
Time taken: 34.453808069229126
{'red': 89, 'blue': 11}
```

```
Time taken: 35.01675581932068
{'red': 84, 'blue': 16}
```

```
Time taken: 37.263699531555176
{'red': 82, 'blue': 18}
```

حال برنامه را به ازای عمق 7 نیز آزمایش میکنیم:

```
Time taken: 644.443213224411
{'red': 87, 'blue': 13}
```

```
Time taken: 617.2342784404755
{'red': 89, 'blue': 11}
```

```
Time taken: 638.6140480041504
{'red': 84, 'blue': 16}
```

یک heuristic خوب چه ویژگی هایی دارد؟ علت انتخاب heuristic شما و دلیل برتری آن نسبت به تعدادی از روش های دیگر را بیان کنید.

هیوریستیک انتخاب شده به این صورت است که فرض میکنیم اگر یک یال جدید به بازی فعلی اضافه شود، چه اتفاقی می افتد. درواقع یکی از ویژگی های هیورستیک خوب نیز همین میباشد که بتواند پیشبینی معقول و درستی داشته باشد. ما به این صورت عمل میکنیم که اگر در حرکت آینده یک یال به شکل اضافه شود، بررسی میکنیم که تشکیل مثلث می دهد یا خیر. اگر مثلث تشکیل شود، بستگی دارد که کدام بازیکن مثلث را تشکیل میدهد. اگر بازیکن آبی مثلث تشکیل دهد، به نفع ماست ولی اگر بازیکن قرمز مثلث تشکیل دهد، به ضرر ماست و طبق همین قضایا امتیاز بندی را در نظر میگیریم. علت انتخاب این هیوریستیک این است که در اکثر مواقع پیشبینی درستی انجام میدهد و احتمال برد بازیکن قرمز را بالا میبرد. از طرف دیگر بر اساس هدف بازی یعنی همان تشکیل مثلث بنا شده است پس نسبت به روش های دیگر برتری دارد.

آیا میان عمق الگوریتم و پارامتر های حساب شده روابطی میبینید؟ به طور کامل بررسی کنید که عمق الگوریتم چه تاثیری بر شانس پیروزی، زمان و گره های دیده شده میگذارد.

بله. طبیعتا هرچه عمق بیشتر باشد، پارامتر های گفته شده نیز بیشتر میشوند. به بررسی هر یک می پردازیم. اگر عمق بیشتر شود، شانس پیروزی افزایش می یابد زیرا میتوانیم تا اینده دورتری را پیشبینی کنیم و حرکت درست تری انجام دهیم که منجر به پیروزی ما شود. اگر عمق بیشتر شود، زمان صرف شده نیز بیشتر میگردد زیرا با افزایش عمق، تعداد حالت ها افزایش می یابد و بررسی آنها زمان بیشتری می طلبد. اگر عمق بیشتر شود، یعنی برای حرکت بعدی، باید خانه های بیشتری را بازدید کنیم پس گره های دیده شده نیز افزایش می یابند.

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 کرده اید؟

این ترتیب اهمیت دارد و ترتیب باید طوری باشد که باعث کاهش زمان اجرای برنامه شود و سرعت ما را افزایش دهد در واقع باید طوری درخت را انتخاب کنیم که تعداد گره های دیده شده کاهش بیاد، اگر در انتخاب درخت به این موارد دقت نکرده باشیم، ممکن است روش هرس کردن فایده ای نداشته باشد و در واقع همان روش بدون هرس اجرا شود.