"بسمه تعالی"

گزارش پروژه دوم هوش مصنوعی – زهرا لطیفی – ۹۹۲۳۰۶۹

سوال ۱: به نظر شما چرا این عامل در بازی اکثر اوقات شکست می‌خورد؟

زیرا Evaluation Function نوشته شده برای آن در هر لحظه تنها به امتیاز خود عامل در State فعلی و بعدی بستگی داشته و بعد هم از میان بیشترین امتیازها، به صورت رندوم یک عمل را انتخاب می‌کند. درواقع اثر اعمال سایر عامل‌ها در این تابع در نظر گرفته نشده است.

```
chosenIndex = random.choice(bestIndices) # Pick randomly among the best
...
return nextGameState.getScore(self.index) -
currentGameState.getScore(self.index)
```

پیاده سازی مینیماکس:

با توجه به شبه کد توضیح داده شده در کلاس پیاده سازی را انجام می‌دهیم. با این تفاوت که ما به تعداد رقبا لایه‌های min داریم و نمی‌توانیم لزوما یکی در میان بین max , min جا به جا شویم. به همین علت لازم است وقتی در عمق min قرار داریم بررسی کنیم عمق بعدی چه نوعی است مربوط به رقیب است یا ما. اگر مربوط به رقیب بود، دوباره min اجرا می‌کنیم در غیر این صورت اگر در نوبت آخرین روح باشیم، max را اجرا می‌کنیم. برای شروع الگوریتم از ریشه هم باتوجه به اینکه همواره ما در ریشه هستیم، کافیست همان تابع max را اجرا کنیم با این تفاوت که حرکت بهینه را در متغیری ذخیره می‌کنیم تا به عنوان جواب برگردانیم.

```
finalChoice = float("-inf")
actions = state.getLegalActions(0)
for action in actions:
    value = self.minChoice(0, 1, state.generateSuccessor(0, action))
    if value > finalChoice:
        finalChoice = value
        nextAction = action
return nextAction
```

حال به طور دقیق‌تر توابع maxChoice و minChoice را شرح می‌دهیم:

در تابع maxChoice ابتدا عمق را به عنوان ورودی می‌گیریم که در ابتدا صفر است و سپس عاملی که قرار است تابع را اجرا کند را مشخص می‌کنیم. (زیرا تابع بازگشتی نوشته شده و هرکدام از max و min یکدیگر را صدا خواهند زد.) انتخاب نهایی را برابر منفی بی‌نهایت قرار داده و اعمال مجاز را فرا می‌خوانیم. سپس عامل اجرا کننده و اکشن درحال اجرا را با state.generateSuccessor فرا می‌خوانیم. این کار باعث می‌شود که بعد از هربار بازی ما، بازی رقیب انجام شود. همین کار برای تمام اکشن‌های مجاز انجام می‌شود و تنها در صورتی مقدار finalChoice را تغییر می‌دهیم که از مقدار قبلی بزرگتر باشد. در این تابع همچنین بررسی می‌شود که آیا به عمق مورد نظر رسیده ایم یا اصلا بازی تمام شده و حرکت مجاز دیگری نمانده است؟ در این صورت self.evaluationFunction بازگردانده می‌شود. در غیر این صورت به صورت بازگشتی، تابع

minChice فراخوانی می‌شود. نوبت به اولین رقیب می‌رسد. در تابع min هربار بررسی می‌شود که آیا تمام رقبا تمام شده‌اند یا خیر. یکی از رقبا بازی خود را انجام داده پس از تعداد عامل‌ها یکی کم می‌کنیم اگر حاصل با عدد agent برابر شد، یعنی رقبا کارشان تمام شده و باید یک عمق بالاتر رفته و تابع ماکس مجدد اجرا شود. در صورت تمام نشدن، تابع min دوباره با عدد agent بزرگتر اجرا می‌شود.

```
def maxChoice(self, depth, agent, state):
    finalChoice = float("-inf")
    actions = state.getLegalActions(agent)
    if depth == self.depth or len(actions) == 0:
        return self.evaluationFunction(state)
    else:
        for action in actions:
            maxValue = self.minChoice(depth, 1,
state.generateSuccessor(agent, action))
            if maxValue > finalChoice:
                finalChoice = maxValue
        return finalChoice

def minChoice(self, depth, agent, state):
    agentNum = state.getNumAgents()
    actions = state.getLegalActions(agent)
    if depth == self.depth or len(actions) == 0:
        return self.evaluationFunction(state)
    else:
        finalChoice = float("inf")
        for action in actions:
            if agent == agentNum - 1 or agentNum == 1:
                minValue = self.maxChoice(depth + 1, 0,
state.generateSuccessor(agent, action))
                if minValue < finalChoice:
                    finalChoice = minValue
            else:
                minValue = self.minChoice(depth, agent + 1,
state.generateSuccessor(agent, action))
                if minValue < finalChoice:
                    finalChoice = minValue
        return finalChoice
```

پیاده سازی هرس آلفا-بتا:

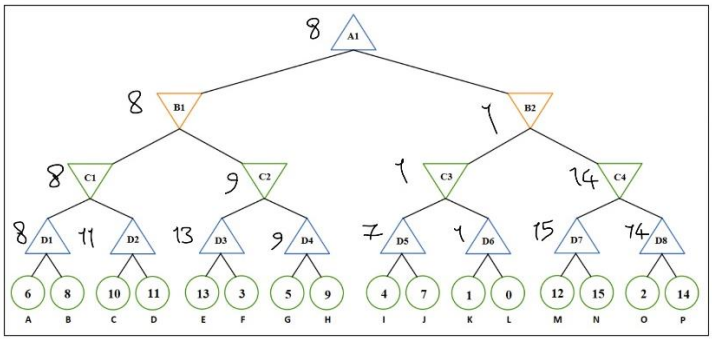
این بخش، کد بهبودیافته همان بخش قبل است. تمامی اعمال قبلی اینجا هم انجام می‌شوند تنها تفاوت در اضافه شدن مقادیر آلفا و بتاست که در ابتدا به ترتیب، منفی و مثبت بی‌نهایتند. این دو متغیر به عنوان ورودی به توابع minChoice و maxChoice داده می‌شوند تا درخت هرس شده و تمام نودها احتیاج به چک کردن نداشته باشند. نتیجتا سرعت این روش از مینیماکس بیشتر خواهد بود. تلاش بر این است که مقدار آلفا را بیشتر کنیم و بتا را کمتر تا جایی که با هم برابر شده یا آلفا بیشتر شود. هرجا این رخ داد، آن گره دیگر احتیاج به بررسی ندارد. در تابع minChoice هر دفعه مقدار را با بتا مقایسه می‌کنیم و بتا را کوچکتر می‌سازیم. در maxChoice هم آلفا را با مقدار مقایسه کرده و آن را بزرگتر می‌کنیم. هرس درخت مینیماکس با آپدیت کردن این دو متغیر تا رسیدنشان به یکدیگر انجام می‌شود. (**بخشی** از کد این بخش در زیر آورده شده.)

```
alpha = float("-inf")
beta = float("inf")
for action in gameState.getLegalActions(0):
    value = self.minChoice(0, 1, gameState.generateSuccessor(0, action),
```

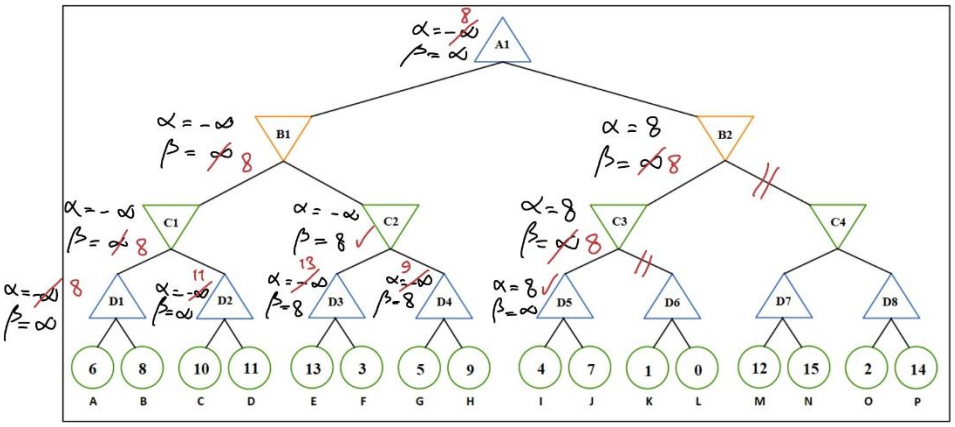
```
alpha, beta)
    if alpha < value:
        alpha = value
        nextAction = action
return nextAction
```

سوال ۲: فرض کنید درخت زیر یکی از تست‌های داده شده به الگوریتم هرس آلفا-بتای شماست. گره‌های مربوط به عامل شما با مثلث آبی و گره‌های دو حریف شما با مثلث برعکس نارنجی رنگ و سبز رنگ نمایش داده شده‌اند .دایره‌ها نیز برگ‌های درخت هستند که حالت‌های A تا P را در آن‌ها ارزیابی کرده و ارزش آن‌ها را داخل این دایره‌ها نوشته‌ایم. الگوریتم مینیماکس خود به همراه هرس آلفا-بتا را برای این درخت اجرا کنید و مشخص کنید که کدام گره‌ها هرس می‌شوند و به چه دلیل. همچنین در پایان مشخص کنید که عامل شما در گره A1 کدام عمل را انتخاب میکند؟

ابتدا بدون اعمال هرس نتیجه را می‌بینیم:



مشاهده می‌کنیم که مینیماکس مسیر منتهی به B1 را انتخاب می‌کند. حال هرس آلفا-بتا را اعمال کرده و مجدد نتیجه را می‌بینیم:



این بار گره‌های D6 و C4 هرس شده اند و در نتیجه آن مقدار گرهای میانی C3 و B2 به ۷ تغییر پیدا می‌کند اما مطابق انتظار، در نتیجه نهایی اثری نداشته و A1 مسیر منتهی به B1 را انتخاب خواهد کرد.

پیاده سازی اکسپکتیماکس:

این بخش هم تغییریافته همان مینیماکس است با این تفاوت که دیگر برای رقبا تابع min نداریم و به جای آن تابع expectimax داریم. درواقع در این تابع میانگین وزنی اکشن‌هایی که هر رقیب می‌تواند انجام دهد را محاسبه کرده و آن را به عنوان عدد اکشن رقیب محاسبه می‌کنیم. به ازای هر رقیب باید این تابع صدا زده شود. بار اول در تابع getAction این کار انجام شده و برای n-1 رقیب دیگر نیز بعد از آن فراخوانی می‌شود. پس از آن نوبت ما می‌رسد و باز تابع max صدا زده می‌شود تا بهترین حرکت را انتخاب کنیم. سپس عمق یک لایه زیاد می‌شود تا جایی که یا به انتهای عمق برسیم و یا بازی تمام شود.

```
def getAction(self, gameState):

    actions = gameState.getLegalActions(0)
    finalChoice = float("-inf")
    for action in actions:
        value = self.expectValue(0, 1, gameState.generateSuccessor(0,
action))
        if value > finalChoice:
            finalChoice = value
            nextAction = action

    return nextAction

...


def expectValue(self, depth, agent, state):
    agentNum = state.getNumAgents()
    actions = state.getLegalActions(agent)
    if depth == self.depth or len(actions) == 0:
        return self.evaluationFunction(state)
    else:
        expect = 0
        for action in actions:
            if agent == agentNum - 1:
                expect += self.maxChoice(depth + 1, 0,
state.generateSuccessor(agent, action))
            else:
                expect += self.expectValue(depth, agent + 1,
state.generateSuccessor(agent, action))
        return expect / len(actions)
```

سوال ۳: چه تفاوتی در عملکرد عامل خود نسبت به عامل آلفا-بتا احساس می‌کنید؟ درباره آن توضیح دهید.

با استفاده از مینیماکس، عامل فرض می‌کند رقبا خصمانه‌ترین حرکت ممکن را انجام می‌دهند به همین دلیل هر تصمیم خود را با این فرض انجام می‌دهد. اما رقبا لزوما خصمانه‌ترین تصمیم را در هر حرکت عملی نمی‌کنند و عامل در هر حرکت این احتمال را هم در نظر می‌گیرد که اگر رقیب خصمانه‌ترین عمل را انجام نداد چه می‌شود؟ و اگر در این صورت به نتایج بهتری رسید، تصمیماتش را بر اساس آن می‌گیرد. درواقع در نوبت‌هایی ریسک کرده و تصمیمات متفاوتی می‌گیرد تا احیانا در حرکات بعدی حالات پر امتیازتری ایجاد شود.

پیاده سازی توابع ارزیابی بهتر:

از میان توابع ارزیابی موجود در مقاله، سه تابع اول را پیاده‌سازی کردیم و جمع وزن‌داری از آن‌ها را به عنوان هیوریستیک نهایی برمی‌گردانیم.

تابع parity:

در این روش، بسته به تعداد رقبا، امتیاز هر عامل محاسبه می‌شود، امتیاز رقبا با هم جمع شده، با امتیاز ما هم جمع شده در صد ضرب شده و بر تفاضل مجموع امتیازهای آن‌ها و امتیاز ما تقسیم می‌شود.

```
# parity
agentNum = currentGameState.getNumAgents()
if agentNum == 2:
    parity = (100 * (currentGameState.getScore(0) -
currentGameState.getScore(1))) / (
                currentGameState.getScore(0) +
currentGameState.getScore(1))
else:
    competitors = sum(currentGameState.getScore(agent) for agent in
range(1, agentNum))
    parity = 100 * (currentGameState.getScore(0) - competitors) /
(currentGameState.getScore(0) + competitors)
```

تابع mobility:

در این روش به جای امتیاز در هر مرحله، تعداد اعمال مجاز برای هر عامل محاسبه می‌شود و در رابطه‌ای همانند رابطه قبلی قرار می‌گیرد.

```
# mobility
agentNum = currentGameState.getNumAgents()
myActions = len(currentGameState.getLegalActions(0))
if agentNum == 2:
    compActions = len(currentGameState.getLegalActions(1))
else:
    compActions = sum(len(currentGameState.getLegalActions(agent)) for
agent in range(1, agentNum))
if myActions + compActions != 0:
    mobility = 100 * (myActions - compActions) / (myActions + compActions)
else:
    mobility = 0
```

تابع corners:

در این روش از تابع currentGameState.getCorners() استفاده می‌شود که یک tuple برمی‌گرداند که ۴ عنصر دارد. هر عنصر نماد یک گوشه است و اگر ۱- باشد، یعنی گوشه ازاد است و در غیر این صورت، شماره عامل را دارد. پس اگر مجموع عناصر این tuple برابر منهای چهار باشد، این هیوریستیک صفر است و در غیر این صورت به اندازه گوشه‌هایی که هر عامل در اختیار دارد به آن امتیاز می‌دهیم و سپس این مقادیر را در همان رابطه روش‌های قبلی قرار می‌دهیم.

```
# corners
add = 0
myUtility = 0
compUtility = 0
for free in currentGameState.getCorners():
    add += free
    value = free
    if value == 0:
        myUtility += 1
    elif value > 0:
        compUtility += 1

if add != -4:
    corners = (100 * (myUtility - compUtility)) / (myUtility + compUtility)
else:
    corners = 0
```

در نهایت با ترکیب هر سه روش با وزن‌هایی که بسته به اثرگذاری حدودی روش انتخاب کردیم داریم:

```
betterEvaluation = (10 * parity) + (200 * corners) + (30 * mobility)
return betterEvaluation
```