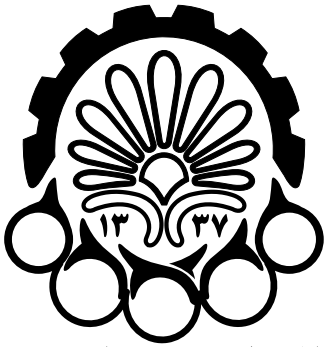

دانشگاه صنعتي اميركبير

(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

گزارش سوم درس هوش مصنوعی

نگارش

آرمان صالحی

استاد راهنما

دکتر مهدی قطعی

بهار 1403

چکیده

گزارش تمرین سـوم (بازی پک‌من) شـامل پیاده‌سـازی یک Simple Reflex Agent به همراه evaluation function آن، و پیاده‌ســازی الگوریتم‌های Minimax و Expetimax به همراه Alpha-Beta Pruning و پیاده‌سازی امتیازی evaluation function مربوط به آن‌ها.

واژه‌های کلیدی:

Simple Reflex Agent – Minimax Algorithm – Expectimax Alghoritm – Atpha-Beta Pruning - Pacman

فهرست مطالب **صفحه**

گزارش کار

پیدا کردن Performance Measure

در ابتدای کار، برای بررسی عملکرد agentمان و تعیین evaluation function مناسب، باید مدل PEAS را در بازی بیابیم. با توجه به واضح بودن environment، sensorها و actuatorها، کافی‌ش صرفا performance measure را به طور دقیق مشخص کنیم. واضح است که performace measure ماکسیمم کردن امتیاز پکمن در انتهای بازی‌ست، اما از آنجا که در صورت پروژه برگلی تحوهٔ امتیازدهی به شکل دقیق مشخص نشده، با گشتن در کد به خطوط مشخص‌کنندهٔ این موضوع در کلاس‌های GhostRule و PacmanRule می‌رسیم.

در ابتدا، با بررسی فایل pacman.py، می‌بینیم که به ازای هر ply کامل در بازی، یک TIME_PENALTY از امتیاز ما کم می‌شود:

```
SCARED_TIME = 40    # Moves ghosts are scared
COLLISION_TOLERANCE = 0.7  # How close ghosts must be to Pacman to kill
TIME_PENALTY = 1  # Number of points lost each round
```

این پنالتی در متد generateSuccessor از کلاس GameState استفاده شده:

```
if agentIndex == 0:
    state.data.scoreChange += -TIME_PENALTY  # Penalty for waiting around
```

که در این کد state.data.scoreChange مقدار تغیر امتیاز بین استیت فعلی و استیت بعدی را نشان می‌دهد.

در مرحلهٔ دوم، با بررسی کلاس PacmanRule به متد زیر می‌رسیم:

```
def consume(position, state):
    x, y = position
    # Eat food
    if state.data.food[x][y]:
        state.data.scoreChange += 10
        state.data.food = state.data.food.copy()
        state.data.food[x][y] = False
        state.data._foodEaten = position
        # TODO: cache numFood?
        numFood = state.getNumFood()
        if numFood == 0 and not state.data._lose:
            state.data.scoreChange += 500
            state.data._win = True
    # Eat capsule
    if(position in state.getCapsules()):
```

```
        state.data.capsules.remove(position)
        state.data._capsuleEaten = position
        # Reset all ghosts' scared timers
        for index in range(1, len(state.data.agentStates)):
            state.data.agentStates[index].scaredTimer = SCARED_TIME
consume = staticmethod(consume)
```

همان‌طور که می‌بینید، با خوردن هر غذا امتیاز ما ۱۰ تا بیشتر می‌شود، از طرفی خوردن آخرین غذا (بردن بازی) ۵۰۰ امتیاز دارد. همانطور اگر ما یکی از کپسول‌ها را بخوریم، امتیازی دریافت نمی‌کنیم ولی Ghostها به اندازهٔ SCARED_TIME به حالت Scared می‌روند. این حالت در مرحلهٔ بعدی بررسی می‌شود.

در مرحلهٔ سوم، با بررسی کلاس GhostRule می‌بینیم که:

```
def collide(state, ghostState, agentIndex):
    if ghostState.scaredTimer > 0:
        state.data.scoreChange += 200
        GhostRules.placeGhost(state, ghostState)
        ghostState.scaredTimer = 0
        # Added for first-person
        state.data._eaten[agentIndex] = True
    else:
        if not state.data._win:
            state.data.scoreChange -= 500
            state.data._lose = True
collide = staticmethod(collide)
```

همان‌طور که می‌بینید، برخورد با ghostها در حالت scared برای ما ۲۰۰ امتیاز مثبت دارد، ولی در حالت معمولی با برخورد به آن‌ها و باخت بازی، ۵۰۰ امتیاز منفی دریافت می‌کنیم.

حال که شمای دقیقی نسبت به شیوهٔ امتیازدهی بازی به دست آوردیم، می‌توانیم به سراغ اولین تسک برویم.

<u>پیاده‌سازی Simple Reflex Agent</u>

با توجه به این که این agent قبلا طراحی شده، ما باید متد evaluationFunction کلاس ReflexAgent را طراحی کنیم. با توجه به خوانا بودن کد، در این‌جا تنها به بررسی ابسترکت کارهای انجام‌شده می‌پردازیم.

برای این کار، ایدهٔ من این است که به تمام آبجکت‌های یک استیت اعم از ghostها، foodها و capsuleها بر اساس فاصله‌شان از پکمن و وضعیت‌شان یک امتیاز اختصاص دهیم. ابتدا تلاش کردم که این امتیاز دهی خطی باشد، یعنی به عنوان مثال، غذایی که با یک حرکت قابل خوردن است ده امتیاز، با دو حرکت نه امتیاز و با سه حرکت هشت امتیاز و ... داشته باشند. به صورت کلی اگر یک چیز در بازی x امتیاز داشته باشد، فاصلهٔ ما از آن d باشد و ماکسیمم فاصلهٔ ممکن d_m باشد، امتیاز نهایی هر آبجت به شکل زیر تعیین می‌شد:

$$scoreWithRespectToDistance = x(1 - \frac{d}{d_m})$$

اما با اجرای این ایده، دیدم که پکمن به جای پیشرفت در بازی و خوردن غذاها، در جای خود می‌ایستاد یا در یک لوپ حرکت می‌کرد. به صورت کلی، پکمن به سمت نزدیک‌ترین غذا نمی‌رفت، حتی اگر فاصلهٔ آن غذا دقیقا یک خانه بود.

علت این اتفاق، این است که اهمیت غذاهای نزدیک به پکمن، به اندازهٔ کافی در این متد زیاد نبود، مثلا ۳ غذا با فاصلهٔ ۲ در این متد ارزش بسیار بیشتری نسبت به یک غذا با فاصلهٔ ۱ دارند. پس من تلاش کردم که این متد را از حالت خطی در بیاورم و نهایتا، متد زیر را نوشتم:

```
def getScoreWithRespectToDistance(initialScore: int, distance: int) -> float:
    """
    Each food, ghost or capsule can have an initialScore, but the distance is
also
    effective in calculating the final score.
    """
    return (initialScore / (distance + 1))
```

این متد با از بین بردن مشکل بالا، باعث شد که امتیازدهی بهتری صورت بگیرد.

نهایتا متد evaluationFunction با دادن امتیاز بر اساس متد بالا به تمامی آبجکت‌های بازی، یک امتیاز نهایی برای هر استیت در نظر می‌گیرد. کد این متد به اندازهٔ کافی گویای نحوهٔ عملکرد آن هست.

با ران کردن تست‌های مربوط به این بخش، می‌بینیم که عملکرد این reflex agent بسیار مورد قبول است و امتیاز کامل تست‌های q1 را می‌گیرد.

```
Average Score: 1401.8
Scores:        1410.0, 1401.0, 1394.0, 1409.0, 1408.0, 1403.0, 1388.0, 1407.0, 1396.0, 1402.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

پیاده‌سازی Minimax

کد این بخش گویای شیوهٔ عملکرد آن است و تلاش شده تا با استفاده از type hintها و اسم‌گذاری درست متغیرها از خوانایی کافی برخوردار باشد. در این الگوریتم، عامل ما با در نظر گرفتن هوشمندی کامل برای تمامی عامل‌های دیگر، به بررسی درخت state space می‌پردازد و حرکت بهینه را انجام می‌دهد. این پیاده‌سازی نمرهٔ کامل تست‌های مربوط به q2 را گرفته است.

با استفاده از کامند زیر، این الگوریتم را با depthهای ۲، ۳ و ۵، برای ۲۰ بار اجرا می‌کنیم:

```
-p MinimaxAgent -l minimaxClassic -a depth=i -n 20 -q
```

خروجی این دستور به شکل زیر است (به ترتیب برای عمق‌های ۲، ۳ و ۵):

```
Average Score: -192.25
Scores:        -492.0, -496.0, 516.0, 516.0, 507.0, -492.0, -492.0, -492.0, -497.0, 516.0, -503.0, 516.0, -497.0, -492.0, -501.0, -497.0, -492.0, 516.0, -492.0, -497.0
Win Rate:      6/20 (0.30)
Record:        Loss, Loss, Win, Win, Win, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss
```

```
Average Score: -142.15
Scores:        -492.0, 513.0, 511.0, -492.0, -495.0, -496.0, -496.0, -496.0, -492.0, -492.0, 510.0, -492.0, 513.0, -492.0, -495.0, -495.0, 511.0, 512.0, 508.0, -496.0
Win Rate:      7/20 (0.35)
Record:        Loss, Win, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Win, Win, Win, Loss
```

```
Average Score: 62.1
Scores:        -492.0, -492.0, 516.0, 516.0, 516.0, -492.0, 516.0, 514.0, 516.0, 516.0, 516.0, -492.0, -494.0, -492.0, -494.0, -492.0, 516.0, 516.0, 516.0, -492.0
Win Rate:      11/20 (0.55)
Record:        Loss, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Loss, Loss, Loss, Loss, Win, Win, Win, Loss
```

همان‌طور که می‌بینید، با افزایش عمق جستجو عملکرد pacman بسیار بهتر می‌شود که این نشان‌دهندهٔ ضعف تابع دیفالت evaluationFunction سوال است. در این تابع می‌بینیم که:

```python
def scoreEvaluationFunction(currentGameState: GameState) -> float:
    """
    This default evaluation function just returns the score of the state.
    The score is the same one displayed in the Pacman GUI.

    This evaluation function is meant for use with adversarial search agents
    (not reflex agents).
    """
    return currentGameState.getScore()
```

یعنی تابع مورد نظر تنها امتیاز همان استیت را برمی‌گرداند و عوامل دیگر را تاثیرگذار نمی‌داند. در بخش نهایی این تمرین، به بهبود این تابع می‌پردازیم.

پکمن با استفاده از این استراتژی، به خوبی زنده می‌ماند (چرا که مرگ را به خوبی تشخیص می‌دهد) ولی خصوصا هنگامی که فاصلهٔ زیادی با غذای باقی‌مانده دارد، از رفتن به سمت آن عاجز است.

پیاده‌سازی Alpha-Beta Pruning

در این مرحله هم کد پیاده‌سازی شده به اندازهٔ کافی خوانا هست و از پیاده‌سازی آلفا-بتای معمولی فاصلهٔ چندانی گرفته نشده. ما در این حالت با در نظر گرفتن هوشمندی کامل برای تمامی agentها در نظر می‌گیریم و با استفاده از این موضوع، به محض این که بفهمیم یک ایجنت هوشمند حرکتی را انجام نمی‌دهد، از بررسی زیردرخت مرتبط با آن حرکت جلوگیری می‌کنیم. در کتاب - Artificial Intelligence A Modern Approach از این استراتژی برای Prune کردن با عنوان Type B Strategy یاد شده:

"A Type B strategy ignores moves that look bad, and follows promising lines “as far as possible.” It explores a *deep but narrow* portion of the tree."

پیاده‌سازی این بخش، نمرهٔ کامل q3 را دریافت کرده است. عملکرد پکمن در این بخش مانند بخش قبلی‌ست چرا که صرفا پرفورمنس الگوریتم را بهتر کرده‌ایم.

پیاده‌سازی Expectimax

در الگوریتم Expectimax، ما با در نظر گرفتن نودهای میانی‌ای به اسم Chance Node برای هر ایجنت[های] حریف، به هر اکشن آن‌ها یک احتمال نسبت داده و بر اساس میانگین وزن‌دار امتیاز استیت‌ها با توجه به این احتمال‌ها، اقدام به حرکت می‌کنیم. مزیت این variation نسبت به Minimax، آن است که ایجنت‌های حریف را تماما هوشمند در نظر نمی‌گیرد و ریسک می‌کند تا امتیاز بهتری داشته باشد.

با توجه به پیچیدگی بیش از حد پیاده‌سازی نرمال Expectimax با استفاده از Chance Nodeها برای مسئلهٔ فعلی، تصمیم گرفتم این الگوریتم را به شکلی متفاوت و ساده‌تر پیاده کنم.

در ابتدا با پیاده‌سازی تابع probabilityOfAction سعی می‌کنیم احتمال هر حرکت حریف در یک استیت از بازی را پیش‌بینی کنیم:

```
def probabilityOfAction(currentGameState: GameState, action: int, agentIdx:
int) -> float:
    legalActions = currentGameState.getLegalActions(agentIdx)

    if action not in legalActions:
        return 0.0

    return 1 / len(currentGameState.getLegalActions(agentIdx))
```

این پیاده‌سازی، با توجه به این که گوست‌ها رندوم حرکت می‌کنند، احتمال یکسانی برای تمامی حرکات برمی‌گرداند.

متد self._pacmanNodeValue مانند یک متد Max-Value در Minimax عادی عمل می‌کند. تفاوت اصلی در متد self._ghostNodeValue است که میانگین وزن‌دار استیت‌ها را برمی‌گرداند:

```
def _ghostNodeValue(self, gameState: GameState, depth: int, agentIdx: int) ->
float:
    if isTerminalState(gameState):
        return gameState.getScore()

    if depth == 0:
        return self.evaluationFunction(gameState)

    weightedAverageValue = 0.0

    for action in gameState.getLegalActions(agentIdx):
        value: float

        if agentIdx == gameState.getNumAgents() - 1:
            value =
self._pacmanNodeValue(gameState.generateSuccessor(agentIdx, action), depth -
1)[0]
        else:
            value =
self._ghostNodeValue(gameState.generateSuccessor(agentIdx, action), depth,
agentIdx + 1)

        weightedAverageValue += value * probabilityOfAction(gameState,
```

```
action, agentIdx)

    return weightedAverageValue
```

این پیاده‌سازی از تست‌های q4 امتیاز کامل گرفته است. در این پیاده‌سازی پکمن ریسک‌پذیرتر است و ممکن است برای غذا به سمت یک روح حرکت کند!

پیاده‌سازی تابعی بهتر برای evaluation (betterEvaluationFunction)

در پیاده‌سازی این تابع، از ایدهٔ تابع evaluation ارائه شده در reflex agent استفاده شده. به جای این که تنها امتیاز پکمن یک استیت را در نظر بگیریم، به همراه آن فاصلهٔ پکمن از هر آبجکت و امتیاز آن آبجکت در بازی، را هم مورد بررسی قرار می‌دهیم. نتیجه، چیزی شبیه به همان تابع قبلی ولی کوتاه‌تر است و با توجه به خوانایی آن، می‌تواند به تنهایی خود را توصیف کند و نیازی به توصیف اضافهٔ آن نیست.

عملکرد این تابع را با استفاده از الگوریتم Minimax، در مپ smallClassic (۲ گوست) و با depth=2 را با ۲۰ بار اجرای آن با عملکرد تابع قبلی مقایسه می‌کنیم:

- عملکرد تابع قبلی

```
Average Score: -269.4
Scores:        -189.0, -119.0, -435.0, -261.0, -805.0, -182.0, -391.0, -597.0, -335.0, -191.0, -170.0, 573.0, -166.0, -234.0, -296.0, -308.0, -266.0, -189.0, -375.0, -452.0
Win Rate:      1/20 (0.05)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

- عملکرد تابع فعلی

```
Average Score: 762.6
Scores:        1550.0, 1124.0, 113.0, 325.0, 1545.0, -403.0, -394.0, 258.0, 1320.0, 1495.0, 1082.0, 1313.0, 1319.0, 499.0, -401.0, 455.0, 1240.0, 323.0, 1164.0, 1325.0
Win Rate:      11/20 (0.55)
Record:        Win, Win, Loss, Loss, Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Loss, Loss, Win, Loss, Win, Win
```

همان‌طور که می‌بینید، تفاوت بسیار زیادی در عملکرد این دو تابع وجود دارد. همچنین این تابع از پس تمام تست‌های q5 با امتیاز میانگینی بالا برمی‌آید و امتیاز کامل این بخش را می‌گیرد:

```
Average Score: 1122.2
Scores:        1292.0, 1267.0, 736.0, 1323.0, 1243.0, 1111.0, 869.0, 1231.0, 938.0, 1212.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```