گزارش پروژه دوم

اشکان شکیبا (9931030)

بخش اول

توضیح کد:

```
food_score = sys.maxsize
for food in newFood.asList():
    distance = manhattanDistance(newPos, food)
    if distance < food_score:
        food_score = float(distance)

ghost_distance_score = 0.0
total_ghost_distances = 0.1
ghosts = successorGameState.getGhostPositions()
for ghost in ghosts:
    ghost_distance = manhattanDistance(ghost, newPos)
    total_ghost_distances += ghost_distance
    if ghost_distance <= 1:
        ghost_distance_score += -1

scared_time_score = sum(newScaredTimes) / len(newScaredTimes)

score = 0.0
score += successorGameState.getScore()
score += 2 * (1 / food_score)
score += -2 * (1 / total_ghost_distances)
score += 5 * ghost_distance_score
score += scared_time_score

return score
```

در متد evaluationFunction، به شکلی برای هر از یک از ویژگی‌های مورد نظر از جمله گرفتن غذاها و دوری از روح‌ها، امتیازی تعریف و محاسبه شده و در نهایت این امتیازات با ضرایبی که بسته به اهمیت آن مورد متفاوت هستند، با هم جمع زده شده و به عنوان حاصل متد بازگردانی می‌شوند.

سوال ۱)

برای هر کدام امتیازی محاسبه می‌شود که می‌تواند مثبت یا منفی باشد.

امتیاز اولیه ضریب ۱، معکوس فاصله نزدیک‌ترین غذا ضریب ۲، مجموعه فاصله روح‌ها ضریب ۲- و تعداد روح‌هایی که بسیار نزدیک هستند ضریب ۵ دارد.

سوال ۲)

می‌توان به پارامتری که افزایش آن ما را از هدف دور می‌کند، ضریب منفی داد.

بخش دوم

توضیح کد:

```
def mini_max(state, iter_count):
    if state.isWin() or state.isLose() or iter_count >= self.depth *
agents_count:
        return self.evaluationFunction(state)
    iter_mod_agents = iter_count % agents_count
    if iter_mod_agents != 0:
        result = sys.maxsize
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
action)
                mini_max_result = mini_max(new_state, iter_count + 1)
                if mini_max_result < result:
                    result = mini_max_result
        return result
    else:
        result = -sys.maxsize
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
action)
                mini_max_result = mini_max(new_state, iter_count + 1)
                if mini_max_result > result:
```

```
                    result = mini_max_result
                if iter_count == 0:
                    action_score.append(result)
        return result
```

ابتدا تابع mini_max بدین شکل پیاده‌سازی شده و سپس با یک بار فراخوانی آن که خود را به شکل بازگشتی فراخوانی می‌کند، امتیاز actionها را محاسبه و action مناسب‌تر را بازگردانی می‌کنیم.

در پیاده‌سازی این تابع ابتدا وضعیت کلی بازی بررسی می‌شود که آیا به پایان رسیده یا به عمق مورد نظر رسیده‌ایم یا نه. در صورتی که هنوز به این حالت نرسیده باشیم، ابتدا با بررسی باقی‌مانده تقسیم iter_count به تعداد عامل‌ها بین پک‌من و روح‌ها تمایز قائل می‌شویم. برای پک‌من ماکسیمم و برای روح‌ها مینیمم بررسی و محاسبه می‌شود و در صورتی که هنوز به پایان نرسیده باشد، با فراخوانی مجدد تابع با عمق بیشتر، فرآیند ادامه می‌یابد.

سوال)

با توجه به اینکه در یک درخت مینیماکس بدترین حالت بررسی می‌شود، با اطمینان یافتن از نتیجه آن بهتر است هر چه زودتر بازی به پایان برسد تا امتیازی که به خاطر زمان از دست می‌دهیم کمینه شود.

بخش سوم

توضیح کد:

```
def alpha_beta(state, iter_count, alpha, beta):
    if state.isWin() or state.isLose() or iter_count >= self.depth *
agents_count:
        return self.evaluationFunction(state)
    iter_mod_agents = iter_count % agents_count
    if iter_mod_agents != 0:
        result = sys.maxsize
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
```

```
action)
                alpha_beta_result = alpha_beta(new_state, iter_count + 1,
alpha, beta)
                if alpha_beta_result < result:
                    result = alpha_beta_result
                if result < beta:
                    beta = result
                if beta < alpha:
                    break
        return result
    else:
        result = -sys.maxsize
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
action)
                alpha_beta_result = alpha_beta(new_state, iter_count + 1,
alpha, beta)
                if alpha_beta_result > result:
                    result = alpha_beta_result
                if result > alpha:
                    alpha = result
                if iter_count == 0:
                    action_score.append(result)
                if beta < alpha:
                    break
        return result
```

ابتدا تابع alpha_beta بدین شکل پیاده‌سازی شده و سپس با یک بار فراخوانی آن که خود را به شکل بازگشتی فراخوانی می‌کند، امتیاز actionها را محاسبه و action مناسب‌تر را بازگردانی می‌کنیم.

پیاده‌سازی این تابع تا حد زیادی مشابه تابع mini_max در بخش دوم است، با این تفاوت که در هنگام بررسی روح‌ها مقدار بتا در نهایت محاسبه می‌شود و در هنگام بررسی پک‌من مقدار آلفا در نهایت محاسبه می‌شود و در هر دوی آنها اگر مقدار بتا از آلفا کمتر شود، حلقه بررسی actionها به پایان می‌رسد و بدین شکل هرس آلفا-بتا صورت می‌پذیرد.

سوال ۱)

مقادیر اعضای درخت پس از اجرای الگوریتم:

a = 8 (alpha = 8, beta = +∞)

```
b1 = 8  (alpha -∞, beta = 8)
b2 = 1  (alpha = 8, beta = 1)
c1 = 8  (alpha = -∞, beta = 8)
c2 = 9  (alpha = -∞, beta = 8)
c3 = 14  (alpha = 8, beta = 14)
c4 = 1  (alpha = 8, beta = 1)
d1 = 11  (alpha = 11, beta = +∞)
d2 = 8  (alpha = 8, beta = 11)
d3 = 13  (alpha = 13, beta = 8)
d4 = 9  (alpha = 9, beta = 8)
d5 = 15  (alpha = 15, beta = +∞)
d6 = 14  (alpha = 14, beta = 15)
d7 = 1  (alpha = 1, beta = 14)
d8 = 4  (alpha = 4, beta = 1)
```

فقط گره P هرس می‌شود، به این دلیل که ماکسیمایزر d8 با مقایسه ۴ و beta که برابر ۱ است، شاخه بعدی را هرس می‌کند.

حرکت بعدی پک‌من نیز به سمت چپ است.

سوال ۲)

در ریشه مقداری متفاوت تولید نمی‌شود، اما در گره‌های میانی ممکن است؛ دلیل این موضوع هم این است که هرس کردن بدون توجه به شاخه‌های بعدی انجام می‌شود و ممکن بود در شاخه‌های بعدی به مقداری بزرگتر (برای ماکسیمایزر) یا کوچک‌تر (برای مینیمایزر) دست یابیم.

بخش چهارم

توضیح کد:

```
def expect_mini_max(state, iter_count):
    if state.isWin() or state.isLose() or iter_count >= self.depth *
agents_count:
        return self.evaluationFunction(state)
    iter_mod_agents = iter_count % agents_count
    if iter_mod_agents != 0:
        scores = []
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
action)
                expect_mini_max_result = expect_mini_max(new_state,
iter_count + 1)
                scores.append(float(expect_mini_max_result))
        result = sum(scores) / len(scores)
        return result
    else:
        result = -sys.maxsize
        for action in state.getLegalActions(iter_mod_agents):
            if action != "Stop":
                new_state = state.generateSuccessor(iter_mod_agents,
action)
                expect_mini_max_result = expect_mini_max(new_state,
iter_count + 1)
                if expect_mini_max_result > result:
                    result = expect_mini_max_result
                if iter_count == 0:
                    action_score.append(result)
        return result
```

ابتدا تابع expect_mini_max بدین شکل پیاده‌سازی شده و سپس با یک بار فراخوانی آن که خود را به شکل بازگشتی فراخوانی می‌کند، امتیاز actionها را محاسبه و action مناسب‌تر را بازگردانی می‌کنیم.

پیاده‌سازی این تابع تا حد زیادی مشابه تابع mini_max در بخش دوم است، با این تفاوت که در هنگام بررسی روح‌ها از امتیازات محاسبه شده برای actionها میانگین گرفته می‌شود و به عنوان حاصل تابع بازگردانی می‌شود.

سوال ۱)

نتیجه مینیماکس (عامل AlphaBetaAgent) همواره باخت در تمام دورهای بازی‌ست، اما نتیجه مینیماکس احتمالی (عامل ExpectimaxAgent)، می‌تواند در بعضی دورهای بازی برد باشد.

این موضوع را می‌توان اینطور توضیح داد که در مینیماکس، ما همواره بدترین حالت را در نظر می‌گیریم و فرض بر این است که حریف به بهینه‌ترین شکل ممکن عمل می‌کند؛ اما زمانی که حریف به شکلی تصادفی عمل می‌کند می‌توان کمی راحت‌تر و با سخت‌گیری کمتر به موضوع پرداخت و در این حالت، مینیماکس احتمالی بهتر می‌تواند شرایط را مدل‌سازی کند.

سوال ۲)

در این الگوریتم احتمال هر حالت بر حسب fitness آن تعیین می‌شود و بعد با انتخاب یک متغیر تصادفی دو حالته، یکی از حالات انتخاب می‌شود. در ادامه می‌توان در هر مرحله با ترکیب کروموزوم‌ها به کروموزوم‌های جدیدی رسید.

به شکل مشابه در بازی پک‌من، می‌توان actionها را به دو حالت صفر و یکی تقسیم‌بندی کرد و متناظر با هر یک از آنها کروموزوم ساخت و با ترکیب کروموزوم‌ها طبق الگوریتم، به پاسخ مطلوب رسید.

بخش پنجم

توضیح کد:

```
food_score = sys.maxsize
for food in foods.asList():
    distance = manhattanDistance(pacmanPosition, food)
```

```
    if distance < food_score:
        food_score = float(distance)

ghost_distance_score = 0.0
total_ghost_distances = 0.1
for ghost in ghostPositions:
    distance = util.manhattanDistance(pacmanPosition, ghost)
    total_ghost_distances += distance
    if distance <= 1:
        ghost_distance_score += -1

scared_time_score = sum(scaredTimers) / len(scaredTimers)

score = 0.0
score += currentGameState.getScore()
score += 2 * (1 / food_score)
score += -2 * (1 / total_ghost_distances)
score += 5 * ghost_distance_score
score += scared_time_score

return score
```

پیاده‌سازی این بخش مشابه بخش اول صورت گرفته است و تنها تفاوت در استفاده از state به جای action است.

سوال)

تابع ارزیابی پیاده‌شده در این بخش تا حد زیادی مشابه بخش اول است و تنها تفاوت آن این است که به جای actionها به بررسی stateها می‌پردازد و به ما این امکان را می‌دهد که با بررسی همه جوانب هر state، چندین گام پس از وضعیت کنونی را بررسی کنیم؛ در حالی که در پیاده‌سازی بخش اول تنها یک گام پیش رو قابل بررسی بود.