فرهاد امان ۹۹۳۱۰۰۶

۱-

```
    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
```

ویژگی‌های مورد نظر ما دور بودن از روح‌ها نزدیک بودن به غذاها میانگین نرخ ترسیدن روح‌ها است.

سوال ۱: برای هرکدام از موارد یک امتیاز را محاسبه می‌کنیم سپس به هر کدام از این امتیازهای ضریبی اختصاص می‌دهیم که با توجه به آن جمع می‌شود. در اینجا امتیاز اولیه ضریب ۱ دارد، معکوس فاصله نزدیک‌ترین غذا ضریب ۲ دارد، مجموعه فاصله روح‌ها ضریب ۲- دارد، و تعداد روح‌هایی که بسیار نزدیک هستند ضریب ۵- دارد.

سوال ۲: کافیست به پارامتری که برای ما ارزش منفی دارد ضریب منفی و برای پارامتری که ارزش مثبت دارد ضریب مثبت داد.

سوال ۳: به طور کلی در Evaluation Function ما حالت فعلی بازی و یک حرکت را به عنوان ورودی می‌دهیم و تابع یک عدد که نشان دهنده میزان مطلوب بودن این حرکت برای ما است را برمی‌گرداند. در نهایت تمام این اعداد با هم مقایسه می‌شوند و حرکتی که بیشترین میزان را دارد انتخاب خواهد شد. باید توجه کنیم که اگر حرکت ما منجر به مرگ پکمن می‌شود امتیاز بسیار پایینی داشته باشد و از ان اجتناب کنیم. اما در عین حال باید کمی فاکتور ریسک هم وارد کنیم در غیر این صورت عامل ما هیچگاه به هدف خود نخواهد رسید. به عنوان مثال در اینجا میزان نزدیک بودن روح‌ها به ما عامل منفی است اما اگر قرار باشد همواره تنها بر اساس همین معیار عمل کنیم هیچگاه نخواهیم توانست تمام غذاها را بخوریم.

۲-

```
def minimax(state, iter_count):
    if state.isWin() or state.isLose() or iter_count >= self.depth * num_agents:
        return self.evaluationFunction(state)
    agent_index = iter_count % num_agents
    if agent_index != 0:
        result = float('inf')
        for action in state.getLegalActions(agent_index):
            if action != 'Stop':
                new_state = state.generateSuccessor(agent_index, action)
                result = min(result, minimax(new_state, iter_count + 1))
        return result
    else:
        result = float('-inf')
        for action in state.getLegalActions(agent_index):
            if action != 'Stop':
                new_state = state.generateSuccessor(agent_index, action)
                result = max(result, minimax(new_state, iter_count + 1))
                if iter_count == 0:
                    action_scores.append(result)
        return result
```

با دریافت وضعیت جاری بازی به عنوان state و شمارنده iter_count که تعداد تکرار فراخوانی تابع را نشان می‌دهد، تابع minimax ابتدا وضعیت پایانی بازی و عمق مورد نظر بررسی می‌کند. اگر به پایان رسیده باشیم یا به عمق مورد نظر رسیده باشیم، تابع امتیاز این وضعیت را با استفاده از تابع evaluationFunction محاسبه می‌کند و آن را باز می‌گرداند.

سپس، تابع agent_index را برای مشخص کردن اینکه این iter_count مربوط به کدام عامل (پک‌من یا روح) است، تعریف می‌کند. اگر agent_index متفاوت از صفر باشد، به دنبال کمترین امتیاز می‌گردد. در این صورت، برای هر عملی که می‌تواند در وضعیت فعلی صورت گیرد، یک وضعیت جدید تولید می‌شود و با فراخوانی بازگشتی تابع minimax برای آن وضعیت، امتیاز حاصل از آن وضعیت محاسبه می‌شود و اگر امتیاز به دست آمده از این وضعیت از حالت فعلی بهتر باشد، آن را به عنوان کمترین امتیاز به‌روزرسانی می‌کند.
اگر agent_index برابر صفر باشد، به دنبال بیشینه امتیاز می‌گردد. در این صورت، برای هر عملی که می‌تواند در وضعیت فعلی صورت گیرد، یک وضعیت جدید تولید می‌شود و با فراخوانی بازگشتی تابع minimax برای آن وضعیت، امتیاز حاصل از آن وضعیت محاسبه می‌شود و اگر امتیاز به دست آمده از حالت فعلی بهتر باشد، آن را به عنوان بیشینه امتیاز به‌روزرسانی می‌کند.

سوال ۱: هنگامی که مطمعن می‌شویم که هیچ راهی برای بردن وجود ندارد باز هم باید بهترین راه را پیدا کنیم در اینجا چون امتیاز بیشتری نمی توانیم بگیریم زمان همچنان مهم است پس هرچه زودتر بازی پایان یابد بهتر است چون در زمان کمتری حداکثر امتیاز را گرفتیم.

سوال ۲: جواب این سوال بالاتر هنگام توضیح کد داده شد.

-۳

```
score_list = []
num_agents = gameState.getNumAgents()

new *
def alpha_beta(state, depth, alpha, beta):
    if state.isWin() or state.isLose() or depth >= self.depth * num_agents:
        return self.evaluationFunction(state)
    current_agent = depth % num_agents
    if current_agent != 0:
        result = sys.maxsize
        for action in state.getLegalActions(current_agent):
            if action != "Stop":
                new_state = state.generateSuccessor(current_agent, action)
                alpha_beta_result = alpha_beta(new_state, depth + 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(current_agent):
            if action != "Stop":
                new_state = state.generateSuccessor(current_agent, action)
                alpha_beta_result = alpha_beta(new_state, depth + 1, alpha, beta)
                if alpha_beta_result > result:
                    result = alpha_beta_result
                if result > alpha:
                    alpha = result
                if depth == 0:
                    score_list.append(result)
                if beta < alpha:
                    break
        return result

alpha_beta(gameState, 0, -sys.maxsize, sys.maxsize)
return [action for action in gameState.getLegalActions(0) if action != "Stop"][
    score_list.index(max(score_list))]
```

تابع alpha_beta در اینجا برای پیدا کردن بهترین عملکرد با استفاده از هرس آلفا بتا استفاده شده است. هرس آلفا بتا یک الگوریتم حریصانه است که در هر مرحله، تنها بهترین مقدار ممکن را برای یک بازیکن در نظر می‌گیرد.

در این تابع، ابتدا با استفاده از state.isWin و state.isLose و همچنین تعداد تکراری رسیدن به اندازه عمق مجاز، اگر باید پایه رسیدن به مرحله بیشتری را بررسی کنیم، بازگشت داده می‌شود. در غیر این صورت، می‌توانیم به عمق بیشتری در بازی حرکت کنیم.

در ادامه، با استفاده از تقسیم بندی در هر مرحله توسط تعداد بازیکنان، مشخص می‌شود که این مرحله به کدام بازیکن اختصاص دارد. اگر مرحله به پک‌من اختصاص داده شود، مقدار آلفا و مقدار بتا به شکل متفاوتی برای او ایجاد می‌شود. سپس با استفاده از حلقه for برای هر یک از حرکت‌های ممکن، یک حالت جدید تولید می‌شود و با استفاده از تابع alpha_beta به دنبال بهترین عملکرد در این حالت می‌گردیم.

در صورتی مرحله به یکی از روح‌ها اختصاص داده شود، کاربرد این تابع تقریباً به همان شکلی است که در بخش دوم برای الگوریتم مینی‌مکس استفاده شده بود، با این تفاوت که مقدار بتا در نهایت محاسبه می‌شود و در هنگام بررسی پک‌من، مقدار آلفا در نهایت محاسبه می‌شود.

سوال ۱:

a = 8 (alpha = 8, beta = inf)

b1 = 8 (alpha -inf, beta = 8)

b2 = 1 (alpha = 8, beta = 1)

c1 = 8 (alpha = -inf, beta = 8)

c2 = 9 (alpha = -inf, beta = 8)

c3 = 14 (alpha = 8, beta = 14)

c4 = 1 (alpha = 8, beta = 1)

d1 = 11 (alpha = 11, beta = +inf)

d2 = 8 (alpha = 8, beta = 11)

d3 = 13 (alpha = 13, beta = 8)

d4 = 9 (alpha = 9, beta = 8)

d5 = 15 (alpha = 15, beta = +inf)

d6 = 14 (alpha = 14, beta = 15)

d7 = 1 (alpha = 1, beta = 14)

d8 = 4 (alpha = 4, beta = 1)

حرکت بعدی پک‌من به سمت چپ است.

سوال ۲: نمی‌تواند در ریشه مقداری متفاوت تولید کند، ولی ممکن است که در گره‌های میانی مقدار متفاوت تولید شود.
در واقع ما در هر مرحله هرس کردن خود را بدون توجه به مقادیر شاخه‌های بعدی انجام می‌دهیم و ممکن است در شاخه‌های بعدی به مقداری بزرگتر یا کوچک‌تر برسیم.

سوال ۳: توضیح کد در بالا انجام شد.

۴-

```
action_scores = []
agents_count = gameState.getNumAgents()

new *
def expectiminimax(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)
                score = expectiminimax(new_state, iter_count + 1)
                scores.append(float(score))
        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)
                score = expectiminimax(new_state, iter_count + 1)
                if score > result:
                    result = score
                if iter_count == 0:
                    action_scores.append(result)
        return result

expectiminimax(gameState, 0)
return [a for a in gameState.getLegalActions(0) if a != "Stop"][
    action_scores.index(max(action_scores))]
```

پیاده سازی این بخش تا حد بسیار زیادی شبیه پیاده‌سازی minimax است تنها تفاوت این است که هنگامی که نوبت روح‌ها فرا می‌رسد به جای اینکه کمینه امتیاز را در نظر بگیریم از امتیازات همه حرکت‌ها میانگین می‌گیریم و از آن میانگین استفاده می‌کنیم.

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.

علت این موضوع این است که در minimax ما همواره بدترین حالت را در نظر می‌گیریم و در واقع به این شکل فکر می‌کنیم که حریف ما همیشه به صورت بهینه بازی خواهد کرد چیزی که با واقعیت تطابق ندارد در نتیجه در حالت expectimax کمی از بدبینی خود می‌کاهیم و به نتایج بهتری خواهیم رسید.

سوال ۲: در این الگوریتم احتمال هر حالت بر اساس برازش آن مشخص می شود و سپس با انتخاب تصادفی یک متغیر دو حالته یکی از حالت‌ها انتخاب می شود. پس از آن، کروموزوم‌های جدید را می توان با ترکیب کروموزوم‌ها در هر مرحله به دست آورد.

به همین ترتیب در بازی پک‌من می‌توان اعمال را به دو حالت 0 و 1 طبقه بندی کرد و متناسب با هر کدام یک کروموزوم ایجاد کرد. با ترکیب کروموزوم‌ها طبق الگوریتم می توان به راه حل مورد نظر دست یافت.

سوال ۳: بهتر است هنگامی که اقدامات ارواح نامشخص و غیر قطعی است از روش Expectimax استفاده شود چرا که در حالت Minimax ما همیشه فرض می‌کنیم که حریف به صورت بهینه عمل می‌کند اما در حالت Expectimax به این شکل است که فرض می‌کنیم حریف از بین حالت‌های قابل انجام برای بازی یکی از حالت‌ها را به صورت تصادفی انتخاب می‌کند.

سوال ۴: پک‌من برای اینکه در این حالت بتواند پیروز شود باید تجربه کند و سپس تجربیات خود را ذخیره کند. به این شکل بعد از مدتی پک‌من پیشرفت خواهد کرد و می‌تواند از آموخته‌های قبلی خود استفاده کند (RL)

۵-

```
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
```

سوال ۱: در این بخش پیاده‌سازی کاملا مشابه بخش اول پروژه است با این تفاوت که به جای action از state استفاده کردیم. استفاده از state به ما اجازه می‌دهد که با بررسی تمام بخش‌های هر state، چندین گام پس از state فعلی را هم در نظر بگیریم در حالی که در پیاده‌سازی بخش اول تنها یک گام پیش رو قابل بررسی بود.