به نام خدا

گزارش کار فاز سوم پروژه اختیاری

MiniMax tree

درس: مبانی و کاربرد هوش مصنوعی

استاد درس: دکتر کارشناس

دستیار آموزشی: پویا صامتی

اعضای تیم :

یونس ایوبی راد - پویا اسفندانی

1 . معرفی پروژه

این پروژه شامل پیاده‌سازی الگوریتم Minimax به همراه بهینه سازی با روش برش آلفا - بتا است. هدف اصلی پروژه، شبیه‌سازی یک بازی ساده است که در آن تصمیم‌گیری هوشمند توسط الگوریتم Minimax انجام می‌شود.

توابع اصلی و الگوریتم های تصمیم گیری در ماژول hen_agent و Bird_agent پیاده سازی شده اند.

تابع get_best_action

```python
def get_best_action(self, grid):

    self.num_action += 1
    best_action = None
    max_eval = -float('inf')
    alpha = -float('inf')
    beta = float('inf')

    for successor, action in AngryGame.generate_hen_successors(grid):
        eval_value = self.minimax(successor, self.max_depth - 1, is_maximizing: False, alpha, beta, self.num_action)
        if eval_value > max_eval:
            max_eval = eval_value
            best_action = action

    return best_action
```

این تابع بهترین حرکت ممکن را برای بازیکن بیشینه‌ساز (Maximizing Player) در بازی انتخاب می‌کند.

1. تعریف مقادیر اولیه:
   - best_action : حرکتی که بالاترین امتیاز را داشته باشد.
   - max_eval : مقدار اولیه ارزیابی، مقدار منفی بی‌نهایت.
   - alphaو beta : مقادیر اولیه برای برش آلفا-بتا.
2. تولید جانشین‌ها:
   - حرکات ممکن برای مرغ (Hen) با استفاده از تابع generate_hen_successors تولید می‌شوند.
   - برای هر جانشین، امتیاز وضعیت بازی با فراخوانی تابع minimaxمحاسبه می‌شود.
3. به‌روزرسانی بهترین حرکت:

- اگر امتیاز محاسبه‌شده بیشتر از max_evalباشد:
  - max_evalبه‌روزرسانی می‌شود.
  - حرکت فعلی به‌عنوان بهترین حرکت (best_action) ذخیره می‌شود.

تابع minimax :

```
def minimax(self, grid, depth, is_maximizing, alpha, beta, num_action):
    if depth == 0 or AngryGame.is_win(grid) or AngryGame.is_lose(grid, num_action):
        return self.evaluate(grid)

    if is_maximizing:
        max_eval = -float('inf')
        for successor, action in AngryGame.generate_hen_successors(grid):
            eval_value = self.minimax(successor, depth - 1, is_maximizing: False, alpha, beta, num_action)
            max_eval = max(max_eval, eval_value)
            alpha = max(alpha, eval_value)
            if beta ⩽ alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for successor, action in AngryGame.generate_queen_successors(grid):
            eval_value = self.minimax(successor, depth - 1, is_maximizing: True, alpha, beta, num_action)
            min_eval = min(min_eval, eval_value)
            beta = min(beta, eval_value)
            if beta ⩽ alpha:  # برش آلفا-بتا
                break
        return min_eval

num_action = 0
```

این تابع برای ارزیابی و تصمیم‌گیری در مورد حرکت بهینه پیاده‌سازی شده است.
ویژگی‌های مهم این تابع شامل موارد زیر است:

- ورودی‌ها:
  - grid : وضعیت فعلی بازی.
  - Depth : عمق جستجوی درخت.
  - is_maximizing : نشان‌دهنده بازیکن فعلی (Max یا Min).

- alpha و beta : مقادیر اولیه برای برش آلفا-بتا.
- num_action : تعداد حرکات انجام‌شده در بازی.

- برش آلفا - بتا:

برای بهبود عملکرد الگوریتم و کاهش تعداد حالت‌های مورد بررسی، از تکنیک برش آلفا-بتا استفاده شده است. این روش بخش‌هایی از درخت جستجو که قطعاً نیازی به بررسی ندارند، حذف می‌کند.

- حالت پایه:

زمانی که یکی از شرایط زیر برقرار باشد، تابع بازگشتی متوقف شده و مقدار ارزیابی برگردانده می‌شود:

- عمق جستجو به صفر برسد.(depth == 0)
- بازی در حالت برد یا باخت قرار گیرد.

ساختار تصمیم‌گیری:

1. بازیکن MAX :

این بخش تلاش می‌کند بالاترین امتیاز ممکن را انتخاب کند و مقدار alphaرا به‌روزرسانی می‌کند.

2. بازیکن MIN :

این بخش پایین‌ترین امتیاز ممکن را انتخاب کرده و مقدار betaرا به‌روزرسانی می‌کند.

تولید جانشین‌ها:

- تابع generate_hen_successors تولید حرکات ممکن برای بازیکن MAX.
- تابع generate_queen_successors تولید حرکات ممکن برای بازیکن MIN.

تابع Evaluate مرغ

یک heuristic برای ارزیابی میزان ارزشمندی و بهتر یا بدتر بودن یک حالت خاص از بازی

```python
def evaluate(self, grid):
    if AngryGame.is_win(grid) and len(AngryGame.get_hen_position(grid)) > 2:
        return -600
    distance_hen = self.find_closet_path_hen(grid)
    distance_egg = self.find_closet_path_egg(grid)
    eggs = AngryGame.get_egg_coordinate(grid)
    if len(eggs) == 0:
        distance_egg = self.find_closet_path_shooter(grid)
        if distance_egg is None:
            distance_egg = -100000
    pigs = AngryGame.get_pig_coordinate(grid)
    hen = math.pow(distance_hen, 1 / 2)
    if distance_egg is None:
        distance_egg = 0
    if hen < 2:
        distance_hen = -1000 * (2 - hen)
    else:
        distance_hen = 0

    return ((self.max_goal_distance - distance_egg) / self.max_goal_distance) * 250 + distance_hen + (
                8 - len(eggs)) * 250 - ((8 - len(pigs)) * 250)
```

بر اساس معیارهای مختلف، امتیازی برای تعیین کیفیت وضعیت گرید (grid) برمی‌گرداند. این امتیاز برای تصمیم‌گیری در الگوریتم Minimax استفاده می‌شود.

معیارهای ارزیابی:

- پیروزی یا شکست :اگر بازی در وضعیت پیروزی باشد و تعداد مرغ‌ها بیشتر از ۲ باشد، امتیاز بسیار منفی (600-) بازگردانده می‌شود.
- فاصله‌ها:
  - فاصله نزدیک‌ترین مرغ (Hen) با دشمن محاسبه می‌شود.
  - فاصله نزدیک‌ترین تخم‌مرغ (Egg) یا نزدیک‌ترین تیرانداز (Shooter) محاسبه می‌شود.
- تعداد آیتم‌ها:
  - تخم‌مرغ‌های باقی‌مانده روی گرید.
  - تعداد خوک‌ها (Pigs) روی گرید.
- تنظیمات وزن‌دهی:
- فاصله مرغ‌ها کمتر از ۲ باشد، امتیاز منفی سنگینی اختصاص داده می‌شود.

- فاصله تخم‌مرغ یا تیرانداز در امتیاز نهایی ضرب شده و تاثیرگذار است.
- تخم‌مرغ‌ها و خوک‌های باقی‌مانده، بر اساس تعدادشان، تاثیر مستقیمی بر امتیاز دارند.

خروجی:

امتیاز نهایی بر اساس ترکیب این معیارها محاسبه شده و به Minimax برگردانده می‌شود تا حرکت بعدی انتخاب شود.

تابع find_closet_path_egg

```python
def find_closet_path_egg(self, grid):
    cloned = copy.deepcopy(grid)
    hen_pos = AngryGame.get_hen_position(grid)
    shooter = AngryGame.get_slingshot_position(cloned)
    if shooter is None:
        return 0
    queen_pos = AngryGame.get_hen_position(grid)
    for i in ((0, 1), (1, 0), (0, -1), (-1, 0)):
        if i[0] + queen_pos[0] not in [-1, 10] and i[1] + queen_pos[1] not in [-1, 10]:
            cloned[i[0] + queen_pos[0]][i[1] + queen_pos[1]] = 'Q'
    cloned[shooter[0]][shooter[1]] = 'R'
    que = Queue()
    que.put((hen_pos, 1))
    while not que.empty():
        index, depth = que.get()
        cloned[index[0]][index[1]] = 'R'
        for i in ((0, 1), (1, 0), (0, -1), (-1, 0)):
            needed_index = [index[0] + i[0], index[1] + i[1]]
            if needed_index[0] in [-1, 10] or needed_index[1] in [-1, 10]:
                pass
            elif cloned[needed_index[0]][needed_index[1]] in ['T', 'Q']:
                que.put((needed_index, depth + 1))
            elif cloned[needed_index[0]][needed_index[1]] == 'E':
                return depth
```

این تابع نزدیک‌ترین مسیر به یک تخم‌مرغ (Egg) را برای مرغ (Hen) در بازی پیدا می‌کند و طول مسیر را برمی‌گرداند.

مراحل اصلی:

1. ایجاد نسخه کپی از گرید: برای جلوگیری از تغییر گرید اصلی.
2. تعیین موقعیت‌ها: موقعیت مرغ، تیرانداز (Slingshot) و تخم‌مرغ شناسایی می‌شود.

3. بررسی مسیرها : با استفاده از جستجوی سطح اول (BFS) و صف، مسیر کوتاه‌ترین فاصله تا تخم‌مرغ پیدا می‌شود.

4. خروجی : طول مسیر کوتاه‌ترین راه) عمق (BFS به تخم‌مرغ بازگردانده می‌شود.

شرایط ویژه:

- اگر تخم‌مرغی پیدا نشود، مقدار پیش‌فرض برمی‌گردد.
- موانع و موقعیت‌ها مانند تیرانداز (R) یا ملکه (Q) در نظر گرفته می‌شوند.

تابع find_closet_path_shooter

```python
def find_closet_path_shooter(self, grid):
    cloned = copy.deepcopy(grid)
    hen_pos = AngryGame.get_hen_position(cloned)
    que = Queue()
    que.put((hen_pos, 1))
    while not que.empty():
        index, depth = que.get()
        cloned[index[0]][index[1]] = 'R'
        for i in ((0, 1), (1, 0), (0, -1), (-1, 0)):
            needed_index = [index[0] + i[0], index[1] + i[1]]
            if needed_index[0] in [-1, 10] or needed_index[1] in [-1, 10]:
                pass
            elif cloned[needed_index[0]][needed_index[1]] == 'T':
                que.put((needed_index, depth + 1))
            elif cloned[needed_index[0]][needed_index[1]] == 'S':
                return depth
```

این تابع کوتاه‌ترین مسیر از موقعیت فعلی مرغ (Hen) به یک تیرانداز (Shooter) را پیدا می‌کند و طول مسیر را برمی‌گرداند.

1. ایجاد کپی از گرید: برای جلوگیری از تغییر گرید اصلی، یک نسخه کپی از گرید ساخته می‌شود.

2. شروع جستجو:
   - موقعیت مرغ (Hen) شناسایی شده و به صف (Queue) اضافه می‌شود.
   - از الگوریتم جستجوی سطح اول (BFS) برای یافتن کوتاه‌ترین مسیر استفاده می‌شود.

3. بررسی همسایه‌ها:

- برای هر موقعیت، همسایه‌ها بررسی می‌شوند.
- اگر موقعیت همسایه یک تیرانداز (S) باشد، عمق فعلی (طول مسیر) برگردانده می‌شود.
- اگر مسیر به مانع (T) یا خارج از گرید برسد، نادیده گرفته می‌شود.

4. پایان:

- اگر تیراندازی پیدا نشود، تابع مقدار پیش‌فرض Noneبازمی‌گرداند.

تابع find_closet_path_hen

```
def find_closet_path_hen(self, grid):
    cloned = copy.deepcopy(grid)
    hen_pos = AngryGame.get_queen_position(cloned)
    shooter = AngryGame.get_slingshot_position(cloned)
    if shooter is None:
        return 0
    cloned[shooter[0]][shooter[1]] = 'R'
    que = Queue()
    que.put((hen_pos, 1))
    while not que.empty():
        index, depth = que.get()
        cloned[index[0]][index[1]] = 'R'
        for i in ((0, 1), (1, 0), (0, -1), (-1, 0)):
            needed_index = [index[0] + i[0], index[1] + i[1]]
            if needed_index[0] in [-1, 10] or needed_index[1] in [-1, 10]:
                pass
            elif cloned[needed_index[0]][needed_index[1]] == 'T':
                que.put((needed_index, depth + 1))
            elif cloned[needed_index[0]][needed_index[1]] == 'H':
                return depth
```

این تابع کوتاه‌ترین مسیر از موقعیت فعلی ملکه (Queen) به یک مرغ (Hen) را پیدا کرده و طول مسیر را بازمی‌گرداند.

1. ایجاد کپی از گرید:

برای جلوگیری از تغییر گرید اصلی، نسخه‌ای کپی ایجاد می‌شود.

2. تعیین موقعیت‌ها:

- موقعیت ملکه (Queen) و تیرانداز (Slingshot) شناسایی می‌شود.

- اگر تیراندازی وجود نداشته باشد، مقدار پیش‌فرض 0بازمی‌گردد.

3. شروع جستجوی سطح اول:(BFS)
   - موقعیت ملکه به صف اضافه می‌شود.
   - همسایه‌های هر موقعیت بررسی می‌شوند:
     - اگر مانعی (T) وجود داشته باشد، جستجو ادامه می‌یابد.
     - اگر یک مرغ (H) پیدا شود، عمق فعلی (طول مسیر) بازگردانده می‌شود.
4. خروجی:
   - اگر مسیری به یک مرغ پیدا شود، طول مسیر برگردانده می‌شود.
   - اگر مسیری وجود نداشته باشد، تابع مقدار پیش‌فرض بازمی‌گرداند.

تابع Evaluate پرنده قرمز

خیلی از توابع Evaluate پرنده قرمز کپی توابع پرنده سفید هستند ولی کمی تفاوت دارد.

```
def evaluate(self, grid):
    eggs = AngryGame.get_egg_coordinate(grid)
    if AngryGame.is_win(grid):
        if self.reward > 1400 or len(eggs) < 3:
            return 1000
        else:
            return -1000
    queen_bird = self.find_closet_path_queen(grid)
    queen_hen,path = self.find_closet_path_hen(grid)
    return  self.calculate(queen_bird)- self.calculate(queen_hen)
```

این تابع وضعیت بازی را ارزیابی می‌کند:

1. **بررسی برد یا باخت:**
   اگر بازیکن شرایط برد (جایزه بالای 1400 یا تخم کمتر از 3) را داشته باشد، 1000بازمی‌گرداند؛ در غیر این صورت، در حالت باخت .1000-
2. **محاسبه مسیرها:**
   نزدیک‌ترین مسیر به پرنده به ملکه (queen bird) و ملکه و مرغ (queen hen) را پیدا می‌کند.
3. **. تابع محاسبه:**
   کارکرد این تابع به گونه ای است که با کاهش مقدار عددی که باز می‌گرداند بیشتر است
4. **مقایسه امتیاز:**
   اختلاف دو مقدار محاسبه‌شده از مسیرها را به‌عنوان امتیاز ارزیابی برمی‌گرداند

نتایج Hard

```
Current Score == 1290
Current Score == 1289
Current Score == 2138
```

نتایج Simple

```
Current Score == 1528
Current Score == 1528
Current Score == 1777
```