

به نام خدا



گزارش کار فاز سوم پروژه

## 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 بهینه‌سازی 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 الگوریتم‌های تصمیم‌گیری در ماژول minimax\_agent.py پیاده‌سازی شده‌اند.

تابع get\_best\_action

```
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.
- alpha و 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 minimax محاسبه می‌شود.

### 3. به‌روزرسانی بهترین حرکت:

○ اگر امتیاز محاسبه شده بیشتر از max\_eval باشد:

▪ max\_eval به روزرسانی می شود.

▪ حرکت فعلی به عنوان بهترین حرکت (best\_action) ذخیره می شود.

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: # بُرش آلفا-بتا
                break
        return min_eval

num_action = 0
```

این تابع برای ارزیابی و تصمیم گیری در مورد حرکت بهینه پیاده سازی شده است.

ویژگی های مهم این تابع شامل موارد زیر است:

• ورودی ها:

○ grid : وضعیت فعلی بازی.

○ Depth : عمق جستجوی درخت.

○ is\_maximizing : نشان دهنده بازیکن فعلی (Max) یا (Min).

○ alpha و beta : مقادیر اولیه برای برش آلفا-بتا.

○ num\_action : تعداد حرکات انجام شده 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:

1. عمق جستجو به صفر برسد. ( $depth == 0$ )

2. 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 alpha را به‌روزرسانی می‌کند.

2. بازیکن MIN :

این بخش پایین‌ترین امتیاز ممکن را انتخاب کرده و مقدار beta را 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

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

```

def evaluate(self, grid):
    if AngryGame.is_win(grid) and len(AngryGame.get_hen_position(grid)) > 2:
        return -600
    distance_hen = self.find_closest_path_hen(grid)
    distance_egg = self.find_closest_path_egg(grid)
    eggs = AngryGame.get_egg_coordinate(grid)
    if len(eggs) == 0:
        distance_egg = self.find_closest_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) برمی گرداند. این امتیاز برای تصمیم گیری در الگوریتم Minimax استفاده می شود.

معیارهای ارزیابی:

- پیروزی یا شکست: اگر بازی در وضعیت پیروزی باشد و تعداد مرغ ها بیشتر از ۲ باشد، امتیاز بسیار منفی (-600) بازگردانده می شود.
- فاصله ها:
  - فاصله نزدیک ترین مرغ (Hen) با دشمن محاسبه می شود.
  - فاصله نزدیک ترین تخم مرغ (Egg) یا نزدیک ترین تیرانداز (Shooter) محاسبه می شود.
- تعداد آیتم ها:
  - تخم مرغ های باقی مانده روی گرید.
  - تعداد خوک ها (Pigs) روی گرید.
- تنظیمات وزن دهی:
- فاصله مرغ ها کمتر از ۲ باشد، امتیاز منفی سنگینی اختصاص داده می شود.

- فاصله تخم مرغ یا تیرانداز در امتیاز نهایی ضرب شده و تاثیرگذار است.
- تخم مرغ ها و خوک های باقی مانده، بر اساس تعدادشان، تاثیر مستقیمی بر امتیاز دارند.

خروجی:

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

تابع `find_closet_path_egg`

```
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 نزدیک ترین مسیر به یک تخم مرغ (Egg) را برای مرغ (Hen) در بازی پیدا می کند و طول مسیر را برمی گرداند.

مراحل اصلی:

1. ایجاد نسخه کپی از گرید: برای جلوگیری از تغییر گرید اصلی.
2. تعیین موقعیت ها: موقعیت مرغ، تیرانداز (Slingshot) و تخم مرغ شناسایی می شود.

3. بررسی مسیرها : با استفاده از جستجوی سطح اول (BFS) و صف، مسیر کوتاهترین فاصله تا تخم مرغ پیدا می شود.

4. خروجی : طول مسیر کوتاهترین راه عمق (BFS) به تخم مرغ بازگردانده می شود.

شرایط ویژه:

- اگر تخم مرغی پیدا نشود، مقدار پیش فرض برمی گردد.
- موانع و موقعیتها مانند تیرانداز (R) یا ملکه (Q) در نظر گرفته می شوند.

تابع find\_closet\_path\_shooter

```
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 می کند و طول مسیر را برمی گرداند.

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

2. شروع جستجو:

○ موقعیت مرغ (Hen) شناسایی شده و به صف (Queue) اضافه می شود.

○ از الگوریتم جستجوی سطح اول (BFS) برای یافتن کوتاهترین مسیر استفاده می شود.

3. بررسی همسایه ها:

- 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 None باز می‌گرداند.

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 باز می‌گرداند.

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 پیش فرض 0 باز می گردد.

3. شروع جستجوی سطح اول: (BFS)

○ موقعیت ملکه به صف اضافه می شود.

○ همسایه های هر موقعیت بررسی می شوند:

▪ اگر مانعی (T) وجود داشته باشد، جستجو ادامه می یابد.

▪ اگر یک مرغ (H) پیدا شود، عمق فعلی (طول مسیر) بازگردانده می شود.

4. خروجی:

○ اگر مسیری به یک مرغ پیدا شود، طول مسیر برگردانده می شود.

○ اگر مسیری وجود نداشته باشد، تابع مقدار پیش فرض باز می گرداند.

نتایج تست بر روی 2 محیط simple , hard

Hard : 1753

```
pygame 2.6.0 (SDL 2.28.4, Python 3.11.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
Score == 1753
```

Simple : 1957

```
pygame 2.6.0 (SDL 2.28.4, Python 3.11.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
Score == 1957
```