دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش پروژه مبانی هوش ـ فاز ۳

استاد:

دکتر حسین کارشناس

دستیار آموزشی:

پوریا صامتی

اعضا گروه:

سپهر فاطمی

پوریا اردستانی

شیما مغزی

دی ۱۴۰۳

فهرست مطالب

عنوان صفحه

۱- الگوریتم Min_Max

۱-۱- پیاده‌سازی تابع Min_Max

تابع min_max با روش هرس کردن آلفا-بتا (Alpha-Beta pruning) پیاده‌سازی شده است.

```
if __name__ == "__main__":

    env = AngryGame(template='simple')

    screen, clock = PygameInit.initialization()
    FPS = 9

    env.reset()
    counter = 0

    running = True
    while running:
        if AngryGame.is_win(env.grid) or AngryGame.is_lose(env.grid,
env.num_actions):
            running = False

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        if counter % 2 == 0:
            _, action = min_max(env.grid, depth=7, is_max=True, alpha=-
math.inf, beta=math.inf,
                                num_actions=env.num_actions)
            if action is not None:
                env.hen_step(action)
            env.render(screen)
            if AngryGame.is_win(env.grid):
                running = False

        if counter % 2 == 1:
            env.queen_step()
            env.render(screen)
            if AngryGame.is_lose(env.grid, env.num_actions):
                running = False

        counter += 1
        pygame.display.flip()
        clock.tick(FPS)
        print(f'Current Score == {AngryGame.calculate_score(env.grid,
env.num_actions)}')

    pygame.quit()
```

همانطور که مشاهده می‌شود، در نوبت بازی hen، برای تعیین کنش مناسب، تابع min_max فراخوانی می‌شود و وضعیت فعلی بازی و عمق پیشروی در درخت به آن داده می‌شود.

تابع min_max:

```
def min_max(grid, depth, is_max, alpha, beta, num_actions):
    if depth == 0 or AngryGame.is_win(grid) or AngryGame.is_lose(grid,
num_actions):
```

```
        return heuristic(grid, num_actions), None
        # return env.calculate_score(grid, num_actions), None

    if is_max:
        max_eval = -math.inf
        best_action = None
        successors = env.generate_hen_successors(grid)
        for successor_grid, action in successors:
            eval_score, _ = min_max(successor_grid, depth - 1, False,
alpha, beta, num_actions + 1)
            if eval_score > max_eval:
                max_eval = eval_score
                best_action = action

            alpha = max(alpha, eval_score)
            if beta <= alpha:
                break
        return max_eval, best_action
    else:
        min_eval = math.inf
        successors = env.generate_queen_successors(grid)
        for successor_grid, _ in successors:
            eval_score, _ = min_max(successor_grid, depth - 1, True, alpha,
beta, num_actions)
            min_eval = min(min_eval, eval_score)
            beta = min(beta, eval_score)
            if beta <= alpha:
                break
        return min_eval, None
```

در ابتدای الگوریتم، ابتدا عمق و وضعیت اتمام بازی بررسی می‌شود تا ببینیم به حالت پایانی رسیدیم یا خیر؛ در صورتی که شرط برقرار باشد، تابع اکتشافی (heuristic) فراخوانی می‌شود تا مقدار ارزیابی حالت فعلی محاسبه شود.

```
if depth == 0 or AngryGame.is_win(grid) or AngryGame.is_lose(grid,
num_actions):
    return heuristic(grid, num_actions), None
```

سپس چک می‌شود که این تابع توسط بازیکن max فراخوانی شده است یا بازیکن min:

```
if is_max:
    max_eval = -math.inf
    best_action = None
    successors = env.generate_hen_successors(grid)
    for successor_grid, action in successors:
        eval_score, _ = min_max(successor_grid, depth - 1, False, alpha,
beta, num_actions + 1)
        if eval_score > max_eval:
            max_eval = eval_score
            best_action = action

        alpha = max(alpha, eval_score)
        if beta <= alpha:
            break
    return max_eval, best_action
```

اگر بازیکن max باشد باید بهترین حرکت ممکن را انتخاب کند که بیشترین امتیاز را برای او دارد. در ابتدا مقدار اولیه تخمین را مساوی با منفی بی‌نهایت می‌گذاریم؛ و تمام حالات بعدی ممکن (Successors) برای هر کنش بازیکن Max تولید می‌شوند.

سپس برای هر یک از حالت‌های ممکن تابع min_max به صورت بازگشتی فراخوانی می‌شود. با قرار دادن is_max=False ، تابع برای نوبت بازیکن Min فراخوانی میشود . به صورت بازگشتی تمام حالات درخت را پیمایش می‌کند. همچنین عمق را ۱ واحد کاهش میدهیم.

مقدار ارزیابی شده و در eval_score و حرکت متناظر محاسبه می‌شود. اگر امتیاز این حالت بهتر از حالت‌های قبلی باشد: بیشترین مقدار (max_eval) را به روزرسانی می‌کنیم و کنش نیز ذخیره می‌شود.

مقدار آلفا به بیشترین امتیاز فعلی بازیکن Max به‌روزرسانی می‌شود. در صورت بیشتر شدن مقدار alpha از beta این حلقه تمام می‌شود در واقع هرس کردن اتفاق می‌افتد. در پایان، بهترین امتیاز و کنش بازگردانده می‌شود.

```
else:
    min_eval = math.inf
    successors = env.generate_queen_successors(grid)
    for successor_grid, _ in successors:
        eval_score, _ = min_max(successor_grid, depth - 1, True, alpha,
beta, num_actions)
        min_eval = min(min_eval, eval_score)
        beta = min(beta, eval_score)
        if beta <= alpha:
            break
    return min_eval, None
```

در نوبت بازیکن Min، مشابه بالا عمل می‌کنیم با این تفاوت که به دنبال به دست آوردن کمترین امتیاز در حین پیمایش هستیم. مقدار اولیه بتا، با مثبت بی‌نهایت انتخاب می‌شود، چرا که به دنبال کمترین مقدار هستیم. تمام حالات بعدی ممکن برای بازیکن Min تولید می‌شود. حلقه برای بررسی هر حالت ادامه می‌یابد و حالات بعدی را ارزیابی می‌کند.

هر حالت (successor_grid) با فراخوانی min_max برای بازیکن Max بررسی می‌شود. اگر امتیاز این حالت کمتر از حالت‌های قبلی باشد کمترین مقدار بروزرسانی می‌شود و مقدار بتا به کمترین امتیاز فعلی بازیکن Min به‌روزرسانی می‌شود. در پایان کمترین امتیاز و کنش بازگردانده می‌شود.

۱-۲- تابع اکتشافی (heuristic)

از تابع heuristic برای ارزیابی و امتیازدهی به state ها استفاده می‌کنیم. این تابع هنگامی که به یک وضعیت پایانی برسیم (بازی win یا lose شود یا به عمق 0 برسیم) اجرا می‌شود.

```
Def heuristic(grid, num_actions):
    hen_pos = env.get_hen_position(grid)
    eggs = env.get_egg_coordinate(grid)
    if AngryGame.is_win(grid):
        if eggs:
            slingshot_distance = 16
        else:
            return 100

    else:
        slingshot_pos = env.get_slingshot_position(grid)
        slingshot_distance = find_nearest_distance(grid, ['B', 'R'],
```

```
slingshot_pos, 'H')

    queen_pos = env.get_queen_position(grid)
    pigs = env.get_pig_coordinate(grid)

    slingshot_distance = find_nearest_distance(grid, ['P', 'R'], hen_pos,
'S')

    queen_distance = find_nearest_distance(grid, ['B', 'R'], queen_pos,
'H')

    queen_factor = 0 if queen_distance > 4 else 0.1
    pigs_factor = len(pigs) * .5
    if eggs:
        dist_to_closest_egg = find_nearest_distance(grid, ['B', 'R', 'Q',
'S'], hen_pos, 'E')
        egg_factor = (10 / dist_to_closest_egg) - (len(eggs) * 10)
        slingshot_factor = 8 - len(eggs)
    else:
        egg_factor = 0
        slingshot_factor = 10
    # queen = -1000 pig = -200 egg=200 goal=400 action =-1
    heuristic_value = (egg_factor + queen_distance * queen_factor +
pigs_factor -
                       num_actions + slingshot_factor / slingshot_distance)
    return heuristic_value
```

با توجه به اینکه می‌خواهیم عامل در ابتدا تخم مرغ هارا بخورد و مستقیم به سمت پرتاب کننده نرود، بررسی می‌کنیم تا در صورت قرار گرفتن در موقعیت برنده شدن و وجود داشتن تخم مرغ در محیط، فاصله پرتاب کننده را به صورت دستی یک عدد بالا می‌دهیم تا مقدار هیوریستیک برای آن کنش کمتر شود.

```
If AngryGame.is_win(grid):
    if eggs:
        slingshot_distance = 16
    else:
        return 100
```

با استفاده از توابع محیط، موقعیت عوامل در محیط را به دست می‌آوریم. و فاصله عامل مرغ را با آنها محاسبه می‌کنیم. برای اینکار از تابع find_nearest_distance استفاده می‌کنیم. این تابع با استفاده از جستجوی BFS، کوتاه‌ترین مسیر تا هدف را برای ما محاسبه می‌کند.

```
def find_nearest_distance(grid, obstacles, start, goal):
    rows, cols = len(grid), len(grid[0])

    def is_valid(x, y):
        return (0 <= x < rows) and (0 <= y < cols) and (grid[x][y] not in
obstacles)


    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left,
Up
    queue = deque([(start[0], start[1], 0)])  # (row, col, distance)
    visited = set()
    visited.add(start)

    while queue:
        x, y, dist = queue.popleft()
```

```
        if grid[x][y] == goal:
            return dist

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if is_valid(nx, ny) and (nx, ny) not in visited:
                visited.add((nx, ny))
                queue.append((nx, ny, dist + 1))

    return 18  # If end is unreachable
```

با استفاده از این تابع فاصله مرغ تا ملکه خوکی، پرتاب کننده، تخم مرغ ها و خوک ها محاسبه می‌شود و این مقادیر با وزن تعیین شده برایشان ضرب شده و مقدار هیوریستیک را تشکیل می‌دهند. برای مثال، برای مدیریت جمع‌آوری تخم مرغ ها به صورت زیر عمل می‌کنیم:

```
eggs = env.get_egg_coordinate(grid)
if eggs:
    dist_to_closest_egg = find_nearest_distance(grid, ['B', 'R', 'Q', 'S'],
hen_pos, 'E')
    egg_factor = (10 / dist_to_closest_egg) - (len(eggs) * 10)
    slingshot_factor = 8 - len(eggs)
else:
    egg_factor = 0
    slingshot_factor = 10
```

اگر تخم مرغی باقی مانده باشد، فاصله مرغ تا نزدیکترین تخم مرغ محاسبه می‌شود؛ و با نزدیکتر شدن مرغ به آن، مقدار egg_factor نیز افزایش می‌یابد تا کنش خوردن آن تخم مرغ، امتیاز هیوریستیک بیشتری داشته باشد. همچنین ضریب slingshot_factor نیز در ابتدا صفر می‌باشد و با کاهش تعداد تخم مرغ‌ها در محیط، این ضریب افزایش می‌یابد تا جمع‌آوری تخم مرغ ها اولویت داشته باشند. اگر تخم مرغی باقی نمانده باشد، ضریب پرتاب کننده افزایش می‌یابد و به هدف اصلی تبدیل می‌شود.

در نهایت مقدار هیوریستیک به صورت زیر محاسبه می‌شود:

```
heuristic_value = (egg_factor + queen_distance * queen_factor + pigs_factor
- num_actions + slingshot_factor / slingshot_distance)
return heuristic_value
```

۱-۳- ارزیابی

امتیازات به دست آمده در محیط‌های simple و hard به صورت زیر می‌باشند:

Simple environment score = 1771

Hard environment score = 1183