인공지능 HW2 보고서

Omok AI

컴퓨터인공지능학부 202323007 이진선

- 목차 -

1. 코드 설명

ai가 오목 수를 두어야 할 때에 act()가 호출되어 바둑돌을 두어야하는 위치를 선정한다.

act()에는 alpha-beta pruning 기법과 minmax 알고리즘을 통해서 각 수의 점수를 평가하고, 가장 효율적인 수를 두도록 만든다.

각 함수들은 act(), alpha_beta(), evaluate(), checking_moves()로 나뉜다.

1) Act()

act() 함수는 ai 수를 세기 위한 집입함수에 해당한다. alpha_beta함수로 들어가기 전의 예외처리와 진입한 뒤의 예외처리 위주로 작성하였다.

게임이 시작된 뒤의 첫 수에는 가중치 판단을 하기 위한 정보가 부족하므로, 무작위 위치를 선택하여 돌을 놓는다. alpha-beta함수로 들어가기 전에 depth를 설정한다. depth는 2로 설정하여, 자신의 차례와 상대방의 차례까지를 각각 고려한다. 탐색의 깊이를 설정하지 않으면 계산량이 많아 timeout이 많이 발생한다. alpha-beta함수를 호출한다.

호출한 뒤에는 예외처리 작업을 한다.
많은 수가 두어지게 되면 배치할 수 있는 곳도 적어지기 때문에 알파베타를 돌고 나서 아무런 곳에도 둘 수 없어 제대로 된 값이 반환되지 않을 수 있다. 해당 경우에는 오류가 발생하기 때문에 if문을 통해서 예외 사항을 처리한다.
해당 경우에는 랜덤한 위치에 배치하도록 작성한다.
최종적으로 x값과 y값을 반환하며 함수가 종료된다.

```python
def act(state : OmokState) :

    #  첫 수를 둘 때에는 연산을 하기 어려우므로 시작점을 랜덤으로 잡기
    if state.num_stones == 0 :
        return np.random.randint(19), np.random.randint(19)

    depth = 2
    player = True
    unuse, move_node = alpha_beta(depth, state, player, -float('inf'), float('inf'))

    if not move_node or move_node[0] is None:
        while True:
            y_pos, x_pos = np.random.randint(19), np.random.randint(19)
            # state에서 생성된 좌표에 돌이 올려져 있는지 여부 체크
            if state.is_valid_position(x_pos, y_pos):
                break
        return y_pos, x_pos

    move_x = move_node[0]
    move_y = move_node[1]

    return move_y, move_x
```

2) Alpha_beta()

minimax 기반의 alpha-beta pruning 알고리즘을 활용한 함수로, 현재의 상태에서 가장 최적의 수를 탐색한다. Alpha와 beta는 각각 현재까지 탐색된 최대값과 최소값을 의미하고, 이 기준들을 통해서 가지치기를 진행하여 불필요한 탐색을 줄이며 성능을 향상시킨다.

Act() 함수에서 alpha_beta() 함수를 초기에 호출할 때에 alpha에는 -inf값을 beta에는 inf값을 전달하여서 얻게된 값에 따라서 갱신한다.

탐색이 종료되는 조건은 두가지로, 설정한 탐색 깊이에 도달한 경우와 바둑판이 가득찬 경우로 나뉘게 된다. 이 경우에는 evaluate()를 통해서 상태들의 점수를 계산한다.

가능한 수는 checking_moves()를 통해서 선발하고 반환된 값은 배치될 수 있는 수들의 리스트로 해당 경우의 수를 탐색하게 된다.

수를 배치할 때에는 단순히 state를 복사하는 것이 아닌 copy.deepcopy()를 활용하였다. 원 상태를 제대로 저장하되 복사한 값을 변경하면서 탐색하기 위함이다.

점수가 beta보다 커졌을 경우에는 탐색할 필요가 없으므로 break를 통해서 가지치기 하고, alpha보다 작아졌을 경우에는 탐색할 필요가 없으므로 break를 통해서 가지치기 한다.

```python
def alpha_beta(depth,  state: OmokState, player, alpha, beta):
    #종료조건이 필요함 - depth가 0이거나, 칸이 모두 찼을 경우
    if depth == 0 or state.num_stones == state.board_size * state.board_size:
        last_x, last_y = state.history[-1]
        my_stone = state.turn
        return evaluate(state, last_x, last_y, my_stone), (0,0)
    #player일 때!
    if player :
        returned_node = tuple()
        check_list = checking_moves(state)
        value = -float('inf')
        for x, y in check_list:
            #상태 복사 필요
            copied_state = copy.deepcopy(state)
            copied_state.update(x, y)

            next_score, unuse = alpha_beta( depth-1, copied_state,False, alpha, beta)

            value = max(value, next_score)
            if value <= next_score :
                value = next_score
                returned_node = (x,y)
            if value > beta :
                break
        return value, returned_node
    #상대방의 다음수를 계산할 때
    else :
        value = float('inf')
        returned_node = tuple()
        check_list = checking_moves(state)
        for x, y in check_list :
            copied_state = copy.deepcopy(state)
            copied_state.update(x, y)
            next_score, unuse = alpha_beta(depth-1, copied_state, True, alpha, beta)
            if value >= next_score :
                value = next_score
                returned_node = (x,y)

            if value >= beta :
                beta = value

            if value < alpha :
                break
        return value, returned_node
```

3) Evaluate()

evaluate()함수는 현재 놓은 바둑돌을 기준으로 게임이 어떻게 구성되어가고 있는지를 평가하고 해당 값을 매기는 평가 함수이다. ai가 수를 둘때에 판단값인 score를 두고 해당 값을 현재의 상태에 따라서 증가, 감소시키는 방식이다. 가로, 세로, 대각선을 구분하기 위해서 4개의 방향에 해당하는 리스트를 만들고, 해당 방향을 for문을 통해서 탐색한다. 각 방향에서 이어져있는 5개의 칸들을 탐색한다. check 리스트에 5개의 연속된 위치 값을 저장하고 탐색을 진행하는데, 보드의

사이즈를 벗어났을 경우에는 none으로 두고, continue를 통해서 평가에서 제외한다. 내가 탐색하는 위치에 자신의 돌만 존재할 때에는 another == 0 이 되고 해당 값은 이길 수 있는 가능성이 높으므로 가중치(*1.3)을 주고 score값에 더한다. 내가 탐색하는 위치에 상대방의 돌만 존재할 경우에는 상대방이 이길 가능성이 높은 패턴이므로 score값에서 제외한다. 이때의 가중치는 공격과 방어에서 어떠한 방식을 우선시 할 건지에 따라서 변경가능하다. 확인하는 라인의 연속되어있는 돌의 개수에 따라서 가중치를 달리한다.

```python
def evaluate(state: OmokState, x: int, y: int, my_stone: int) :
    score = 0
    dir = [ (1, 0), (0, 1), (1, 1), (1, -1) ]

    for dx, dy in dir:
        for history in range(-5, 5):
            check = []
            end = 0
            for i in range(5):
                nx = x + (history + i) * dx
                ny = y + (history + i) * dy
                if 0 <= nx < 19 and 0 <= ny < 19:
                    check.append(state.game_board[ny][nx])
                else:
                    check.append(None)
                    end += 1

            if check.count(None) > 0:
                continue

            count = check.count(my_stone)
            another = check.count(-my_stone)

            if another == 0:
                # 방해 없는 순수한 연속돌
                get_score = {1: 8, 2: 80, 3: 800, 4: 80000, 5: 1000000}
                score += get_score.get(count, 0)*1.3

            elif count == 0:
                # 방어 고려: 상대방 연속
                minus_score = {1: 8, 2: 80, 3: 800, 4: 80000, 5: 1000000}
                score -= minus_score.get(another, 0)*0.8
    return score

```

4) Checking_moves()

checking_moves는 탐색할 후보 자리에 대해서 탐색하는 함수로 후보자리가 많을 경우에 탐색을 할 때에 시간이 많이 소요되므로 만들게 된 함수이다.

최근에 둔 수를 기준으로 주변 공간을 탐색하면서 둘 수 있는 공간을 후보로 담

아둔다.

state.history[-4:]를 통해서 최근에 두어진 수 4개를 탐색하게 되고 해당 공간을 기준으로 7x7 공간을 탐색하면서 아직 수가 놓여지지 않은 곳을 후보로 추가한다.

탐색하는 공간에 후보가 아예 없을 경우에는 보드 전체를 순회하면서 빈칸을 모두 후보에 추가하게 된다.

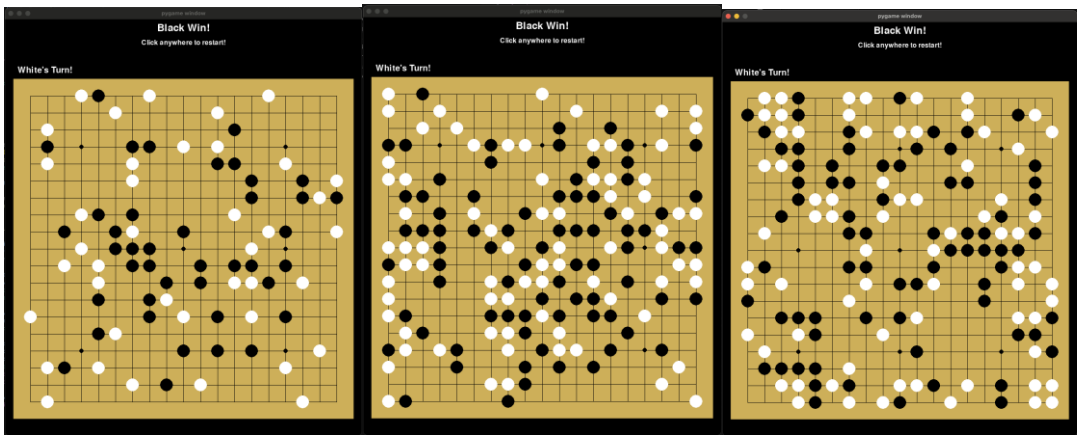
```python
def checking_moves(state : OmokState):
    size = state.board_size
    board = state.game_board
    moves = set()

    for x, y in state.history[-4:]:
        for dy in range(-3, 4):
            for dx in range(-3, 4):
                nx = x + dx
                ny = y + dy
                if 0 <= ny < size and 0 <= nx < size :
                    if board[ny][nx] == 0 :
                        moves.add((nx, ny))

    if not moves :
        for y in range(size):
            for x in range(size):
                if board[y][x] == 0:
                    moves.add((x, y))

    return moves
```

2. 실행 결과 분석



실행 결과 분석을 위한 실험은 랜덤하게 배치하는 상대코드와 겨루어 진행했다.

플레이어의 의도를 파악하기 위해서 플레이어가 둔 위치를 중심으로 탐색을 진행하며 해당 위치에서의 깊이 2를 탐색을 하게 되는 방식으로 하기 때문에 전체를 탐색하는 방식보다 효율적이다.

해당 코드에 대해서 여러 실행을 진행해보았지만 timeout은 발생하지 않았다.

최근 4수를 기준으로 총 7x7칸을 탐색하므로 최대 196칸(49x4)을 탐색하게 된다.

100~200개의 후보군에서 깊이 2만큼 탐색하기 때문에 시간적으로 안정적이다.

evaluate함수는 5칸씩 슬라이딩하면서 점수를 계산하는데 해당 값에 대해서 추가적인 연산없이 갯수를 통해서 점수를 매기기 때문에 각각의 점수를 매기는 데에 빠르게 연산할 수 있다.

문제가 되는 예외처리에 대해서 대처할 수 있는 방안을 마련해두었으므로 오류는 최대한으로 발생하지 않도록 하였다.

랜덤 배치하는 코드를 활용하여 ai_agent.py를 활용하였을 때에는, 모든 상황에서 user_agent.py가 승리했다.

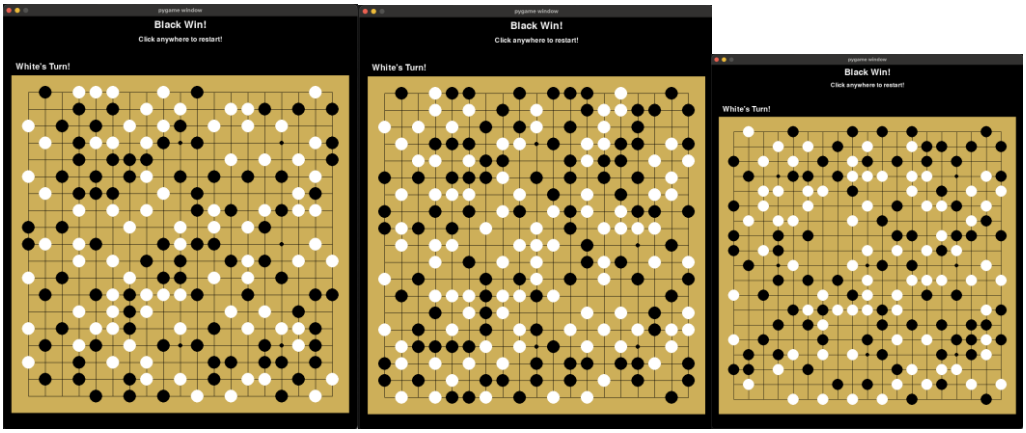
또한 랜덤으로 서치하는 ai_agent.py보다 더욱 빨리 배치하는 경우도 많았다.

따라서 시간적인 부분과 전략적인 부분에서 안정적이라고 평가할 수 있었다.

복잡하게 계산하는 방식이 아니기 때문에 전략적인 수에 대해서는 약할 수 있으나 방어적인 전략으로 인해 상대방이 점수를 얻는 것에 있어서 안정적으로 방어할 수 있었다.

3. 성능 비교

    1) AI vs AI (동일 코드로 작동시켰을 경우)



두 플레이어 모두 동일한 AI 코드를 활용하여 게임을 진행하였다.

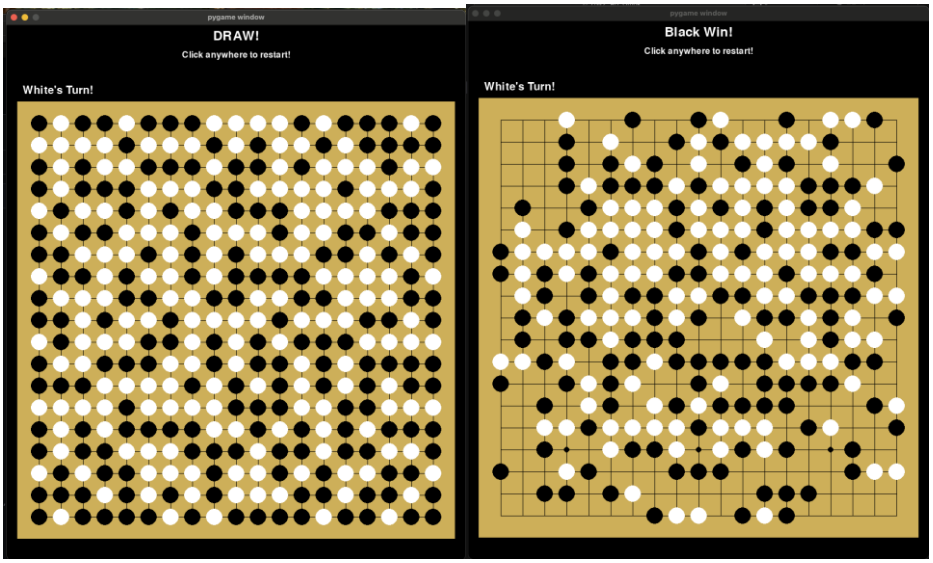
실험을 5번 이상 실행한 결과, 모든 게임이 검은 돌의 승리로 종료되었다.

Alpha-beta pruning 탐색 중에 배치할 수 있는 위치가 없거나 평가가 불가능한 경우에 move_node가 빈 튜플로 반환되는 경우가 존재했고, 해당 경우에는 move_x에서 move_node로의 접근을 할 때에 오류가 발생하는 문제점이 있었다. 해당 문제점을 개선하기 위해서 if not move_node or move_node[0] is None: 을 통해서 예외처리를 진행하도록 코드를 수정하였다.

이 해결방법을 통해서 모든 상황에서 ai가 놓을 수 있는 위치를 탐색할 수 있게 되었고 게임은 정상적으로 종료되었다. 대부분의 경우에서 서로의 수를 방어하는 방어적인 플레이가 반복되는 문제점을 찾을 수 있었다.

공격보다는 방어에 집중하는 문제점으로 인해서 승부를 내는 계산에서는 부족한 모습을 보였다.

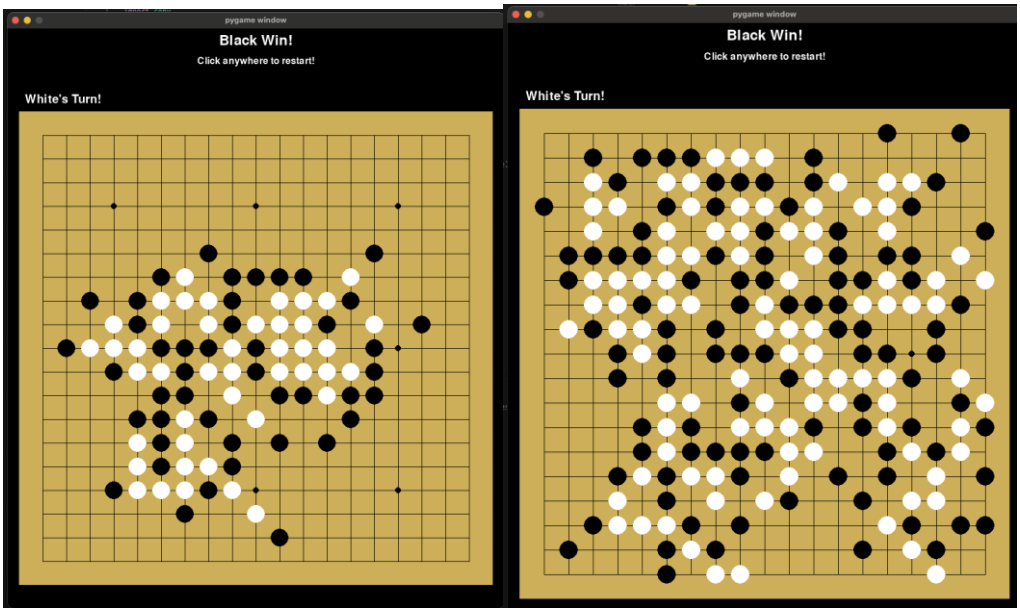
2) AI vs other AI (해당 코드와 다른 ai 코드로 작동시켰을 경우에)



직접 개발한 AI와 웹에서 제공되는 오목 게임을 겨루어 게임을 진행하였다. State.human = True로 두고 사람이 직접 수를 두어 외부 ai에서 나온 결과를 직접 입력하는 방법과 다른 사람이 제작한 오목 게임을 활용하여 state.human = false로 두고 진행하는 방법과 같이 두가지 방법을 활용하였다.

대부분의 게임에서는 대다수 무승부로 종료되거나 일부 이기는 경우가 있었다. 이기는 경우가 발생하는 경우에는 수 계산에서 이겼다기 보다는 제작한 ai 는 제한 시간 내에 외부 ai는 제한 시간 내에 배치하지 못해 랜덤하게 배치되어 승률이 달라진 경우에 해당했다. 외부의 수는 대체로 공격적인 수로 게임을 진행하는 반면에 제작한 코드는 방어적인 수로 진행하여 상대방의 수를 무력시키는 방식으로 진행되었다. 최대한 상대방의 수를 무력시키면서도 자신의 수를 얻어갈 수 있는 방법을 선택하기 때문에 효율적인 수가 생성되게 되었다.

3) 사람 vs AI



사람과 경쟁을 하는 경우에는 대체로 AI가 이겼다. 이는 aI가 사람이 둔 수를 인식하고 해당 수를 대응하는 수로 이루어져있어서 줄을 세우려고 시도하지만, 차단되어 연속적으로 놓을 수 없는 상황으로 인해서 발생했다. 이러한 상황이 지속됨에 따라서 사람은 자신이 두는 수에 대해서 집중하는 경우가 많아졌고, ai는 사람의 수를 방어하면서 동시에 자신에게 유리한 수를 두었다. 이러한 상황이 반복됨에 따라서 ai 가 우위를 차지했다.

4. 문제점 분석 및 개선 방안

지금까지 작동하는 AI의 방향성을 파악하고 문제점을 분석하였을 때에 최대 3가지로 문제점을, 나눠서 볼 수 있었다.

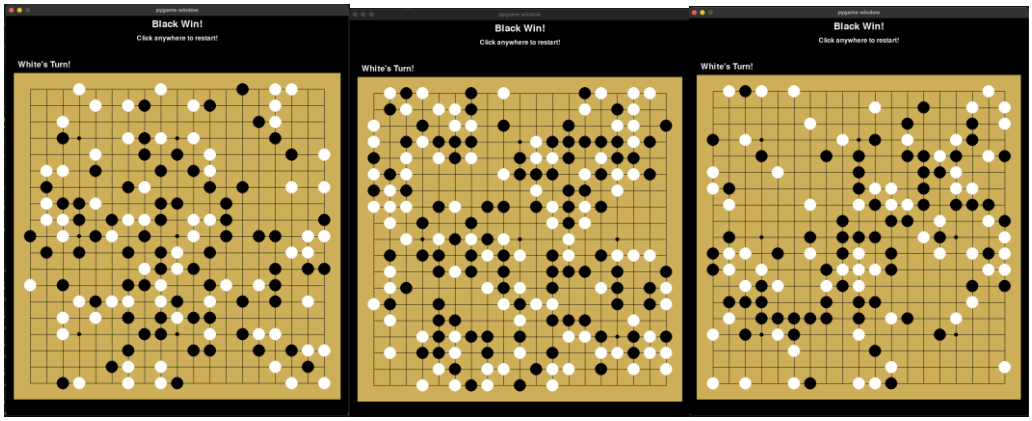
첫번째, 수를 분석하는 데에 너무 방어적인 부분에 치우쳐 있다는 부분이 문제점으로 파악되었다. 두 번째, depth가 2로 현재 자신의 수와 상대방의 수, 두개밖에 파악하지 못하여 승리를 앞에 두었을 때 제대로 분석하지 못하는 점이 있었다. 세번째, 가중치의 비율이 임의로 선정한 것이기 때문에 아쉬운 점이 존재했다.

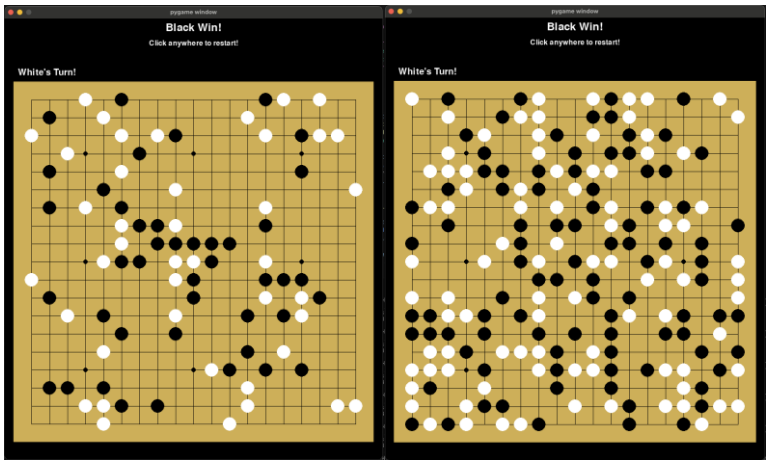
첫번째 문제에 대해서는 수의 가중치를 변화시키는 방법으로 개선시켜보았다.

두번째 문제에 대해서는 depth를 3,4로 증가시켜 실험을 진행해보았다.

세번째 문제에 대해서는 가중치에 대한 논문을 찾아서 점수를 재배치해보았다.

1) 가중치 조절





해당 분석은 상대방은 random으로 배치했을 때를 가정하여 ai의 evaluate함수 내의 공격에서의 가중치를 증가시키며 성능 비교를 진행하였다.

사진의 순서대로 *1.3, *3, *5, *10, *20 배 로 변경하여 실험을 진행하였다.

Evaluate함수에서는 score+= get_score.get(count, 0) *[배수]에서 배수를 변화시키면 실험을 진행하였다.

Score-=minus_score.get(another,0)*0.8으로 방어 가중치는 고정한 상태에서 진행하였다.

1.3배에서 예상보다 안정적인 수를 두는 모습을 보여주었다. 공격의 가중치와 방어의 가중치가 안정적으로 비교되었기 때문으로 예상된다. 그럼에도 불구하고 상대방의 수보다는 자신의 수에 집중하지 못하며 놓을 수 있는 수가 있음에도 놓지 못하는 모습을 보여주기도 했다. 3배에서는 점수를 얻을 수 있는 기회를 많이 확보했지만 기회가 왔음에도 방어에 집중하느라 해 위치에 배치하지 못하는 문제점이 존재했다. 점차 배수를 늘려갈수록 상대방의 수보다 자신의 수에 집중하며 공격적으로 수를 두는 모습을 보였다. 20배로 실험을 진행했을 때에는 처음 수부터 자신의 수를 늘려가는 공격적인 모습을 보여주기도 했다. 따라서 공격의 가중치를 증가시킬수록 방어적인

수도 있지만, 공격적인 수를 선택하는 것을 알 수 있었다.

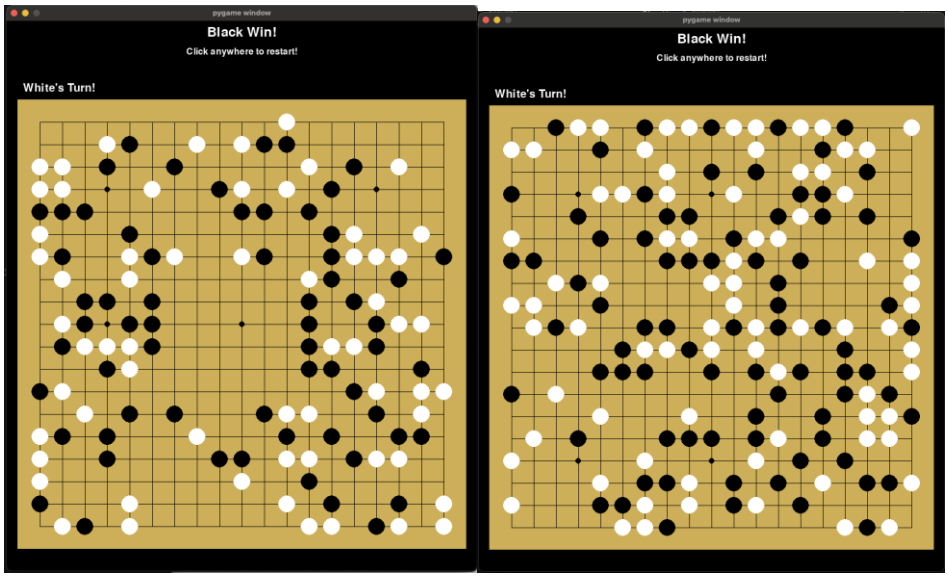
2) depth의 크기에 따른 변화

Depth를 3으로 증가시키니 바로 timeout이 발생해서 수가 제대로 두어지지 않는 문제점이 발생했다. 해당 문제에 대해서 3, 4로 증가시켜 실험을 진행해볼 예정이였으나 depth가 3으로 증가시키자마자 timeout으로 인해서 제대로 수가 두어지지 않아 depth 증가시켰을 때에 탐색이 매우 느려졌음을 알 수 있었다. Time을 증가시켰을 때에는 탐색이 가능하긴 했지만 depth가 2였을 때와 계산적인 부분에서 큰 차이는 존재하지 않았다.

이는 depth가 4일 때도 동일한 문제점이 존재했다.

특히 random으로 두어지는 상대이기 때문에 상대방의 수를 예측하면서 최적의 수를 두는 것이 큰 의미를 갖지 못했다.

3) 연속된 수에 따른 가중치 변화



해당 실험은 ai가 연속된 돌의 수에 따라서 평가하는 점수 자체를 조정하였을 대에 전반적인 수 선택과 성능에서 얼마나 영향을 미치는지에 대해서 분석하기 위해서 진행되었다.

가중치를 {1: 8, 2: 80, 3: 800, 4: 8000 ,5: 10000000} 에서 {1: 8, 2: 80, 3: 800, 4: 80000 ,5:

100000000} 로 증가시켜서 실험을 진행했다. 두 사진 모두 증가시켰을 때의 결과이다. 해당 실험은 랜덤하게 배치하는 상대와 실험을 진행했다.

해당 사진을 보았을 때에 가중치를 변화시키기 전과 후로 비교하였다.

가중치를 변화시키기 전 실험 결과는 실행 결과 분석에 있는 실험결과와 비교하여 진행했다.

해당 가중치를 변화시키기 전에는 랜덤하게 이동하는 흰 돌을 따라서 수를 분석하느라 자신의 수를 제대로 살피지 못하는 모습을 볼 수 있었다. 따라서 많은 수를 놓은 후에야 이길 수 있었던 문제점이 있었다.

하지만 가중치를 증가시킨 후에는 4개와 5개가 이루어진 수에 대해서 크게 반응하고 해당 점수를 중요도에 있어서 크게 가져가는 모습을 볼 수 있었다.

4개가 놓일 수 있는 가능성과 5개가 놓일 수 있는 가능성에 대해서 큰 값을 가지고 가는 모습을 보고 값의 크기에 따라서 성능의 차이가 큼을 파악할 수 있었다.

5. 결론

제작한 오목 AI는 수를 계산할 때 공격과 방어 상황을 동일한 방식으로 평가하게 될 경우에 대체로 방어적인 수를 우선시하는 경향이 나타나서 점수를 낼 수 있는 상황임에도 불구하게 제대로 점수를 내지 못하는 상황이 발생했다. 이로 인해서 게임이 무의미하게 길어지거나 무승부로 종료되는 경우가 많았다. 이를 개선하기 위해 공격 시 얻는 점수의 가중치를 높이고, 방어 상황에 대한 점수는 낮게 설정하는 방식으로 evaluate() 함수를 수정하였다. 그렇게 여러 가지 테스트를 통해서 가장 효과적인 공격과 수비의 가중치 비율 선정이 가능했다. 하지만 현재 evaluate() 함수는 연속된 돌의 개수를 통해서 평가를 하는 방식으로 구성되어 있어, 다양한 수에 대해서 계산하지 못하는 한계가 존재했다. 이러한 부분은 방어적인 부분에 대해서는 우수할 수 있으나 실질적으로 점수는 내는 부분에서는 아쉬운 부분이였다.

더하여 실험을 진행해보았을 때, 상대가 랜덤으로 수를 두는 경우와 계산 후 수를 두는 경우에 AI가 가져갈 수 있는 행동이 서로 다른 경우에 효과적이라는 것을 알게 되었다.

이러한 부분에서는 상대방의 상태에 따라서 다르게 반응할 수 있는 AI 혹은 어떠한 상황에서도 안정적으로 대응할 수 있는 AI가 적절하다고 판단되었다.

공격적인 부분보다 방어적인 부분이 우세하다는 점에서 아쉬운 점을 가지고 있었지만 어떠한 상황에서도 안정적으로 대응할 수 있었다는 부분에서 이점을 가지고 갈 수 있다고 분석하게 되었다.