202312807 정사임 hw2 보고서

이 보고서는 오목 게임에서 검은 돌을 이기는 코드를 구현하며 게임 트리 및 Alpha-Beta Pruning(가지치기)를 구현하는 것이 목표이다.

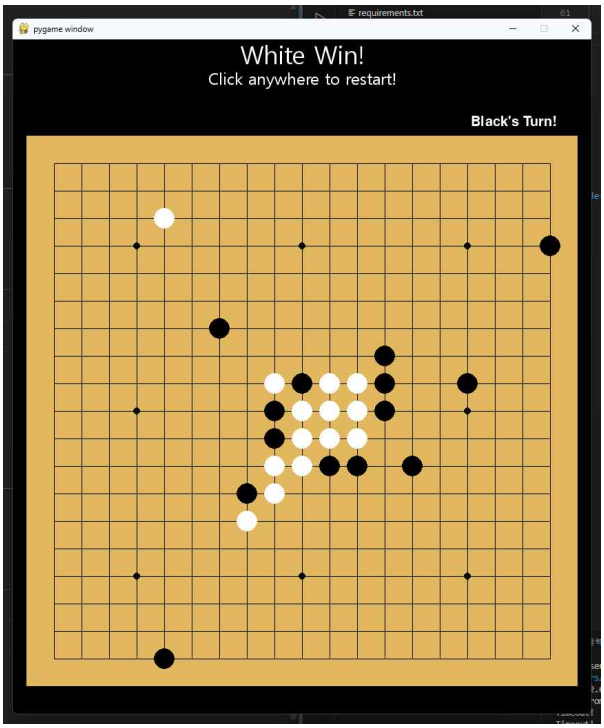
Ⅰ. 코드 분석

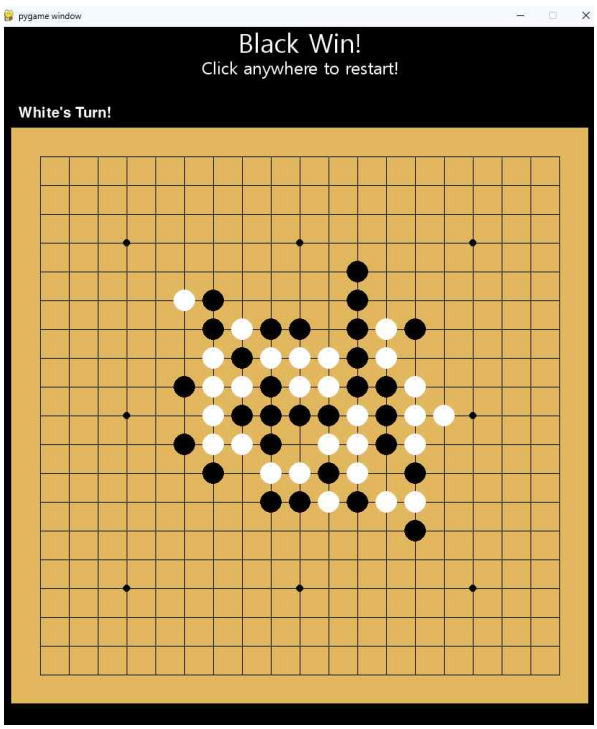
❶. act() 함수 : ai가 다음 수를 고르는 것을 책임지는 역할을 한다.
- 흑이 다음 수에서 승리할 경우 바로 두기
- 백이 다음에 이기면서 승리할 경우 차단
- 백이 생성할 가능성이 큰 4목 혹은 양쪽이 열린 3목이 되는 경우 차단
- 위에 해당하지 않는 상황이면 Alpha-Beta Pruning을 통해 가장 효율적인 경우를 탐색
- timeout이면 랜덤으로 찾음

❷. alpha-beta() 함수 : 가장 괜찮은 수의 탐색을 위해 minimax 탐색에 Alpha-Beta Pruning을 적용하여 가지치기를 수행한다.(불필요한 가지를 줄인다.) 더 이상 탐색할 필요가 없을 경우 조기에 탐색을 종료한다. 제한된 시간 안에 가능한 깊이까지 탐색하여 최적의 수를 선택한다.

❸. evaluate() 함수 : 현재 보드 상태를 평가하여 점수를 반환한다. AI가 이긴 경우 매우 높은 점수를, 상대가 유리한 경우에는 마이너스 점수를 부여한다.

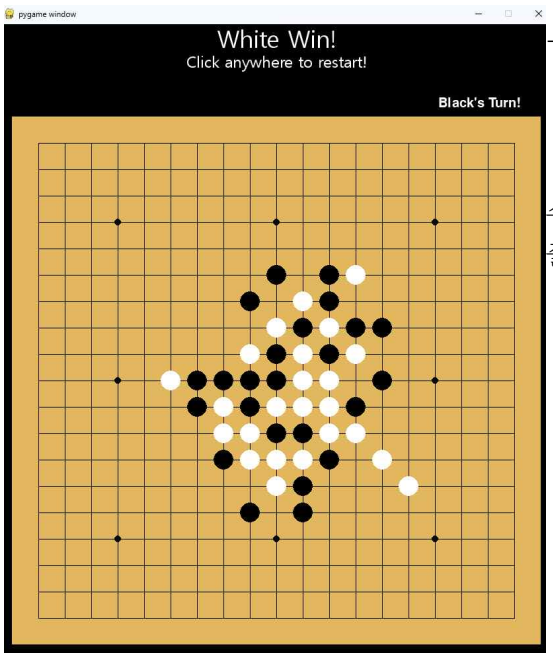
Ⅱ. 결과 분석



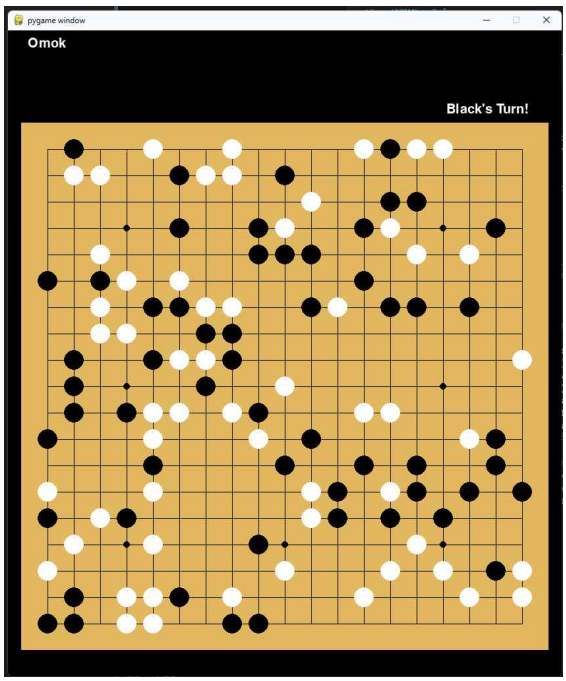


-> human = true, Max_depth = 3, time limit: 4.8s

사용자(백)와 ai(흑)이 play한 모습이다.
흑은 백이 3목 이상일 때 수비를 할 수 있도록 설정하였다.
이러한 조건을 설정하였음에도 왼쪽 사진에서 백이 이길 수 있었던 이유는?
-Ai는 가장 위험한 수 하나만 막도록 되어 있어서, 다른 방향의 위협을 놓치는 경우가 생길 수 있다. evaluate()의 점수가 낮은 쪽은 막지 않고, 더 위험하다 판단된 쪽을 막는 것
-수비 타이밍 놓친 경우
▶점수(흑)
5목: +100000
4목: +10000
3목: +1000
2목: +200
백이 위의 형태-> 각각 -30000, -5000등으로 처리한다.



-> 이 결과는 main에 있는 time을 10초, user_agent에 있는 time을 8초로 설정했을 경우이다.
AI가 더 깊은 수까지 탐색할 수 있어 똑똑한 수를 둘 수 있고, 시간이 넉넉하여 timeout을 줄일 수 있다는 장점이 있었다.



-> human = false인 경우
user agent vs ai agent (랜덤 ai)

초기에 실행했던 코드는 흑과 백이 서로 3목 이상일 경우에 수비를 하지 않고, 랜덤한 위치에 수를 놓아 게임이 끝나는 데 걸리는 시간이 매우 길었다.
ai_agent로 돌릴 경우 백은 전략이 전혀 없는 상태이기 때문에 이길 기회가 와도 놓치고, 흑은 백이 아무 위치에 수를 두기에 열려있는 곳이 너무 많아 수를 엉뚱한 곳에 놓게 된 것이다.
-> 먼저 흑이 너무 떨어지지 않은 위치에(서로 모여서) 수를 두도록 설정하였다.

초기 코드:

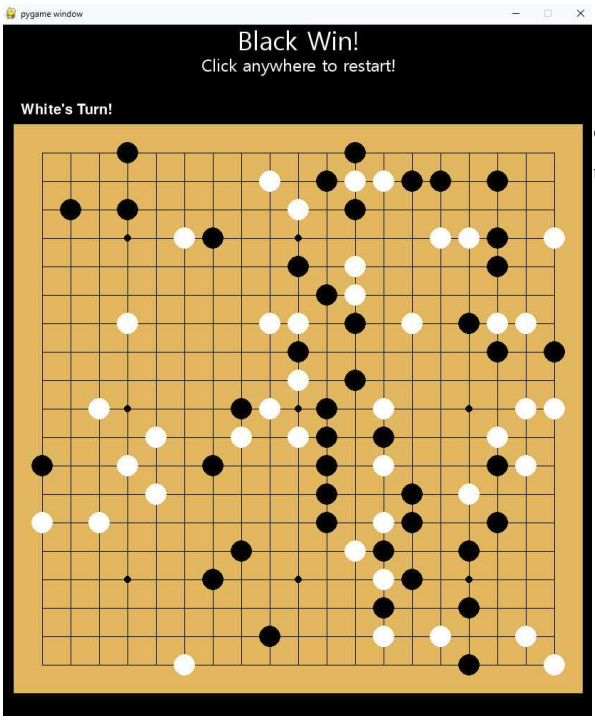
```
for y in range(state.board_size):
    for x in range(state.board_size):
        if state.is_valid_position(x, y):
            moves.append((y, x))
```

개선한 코드:

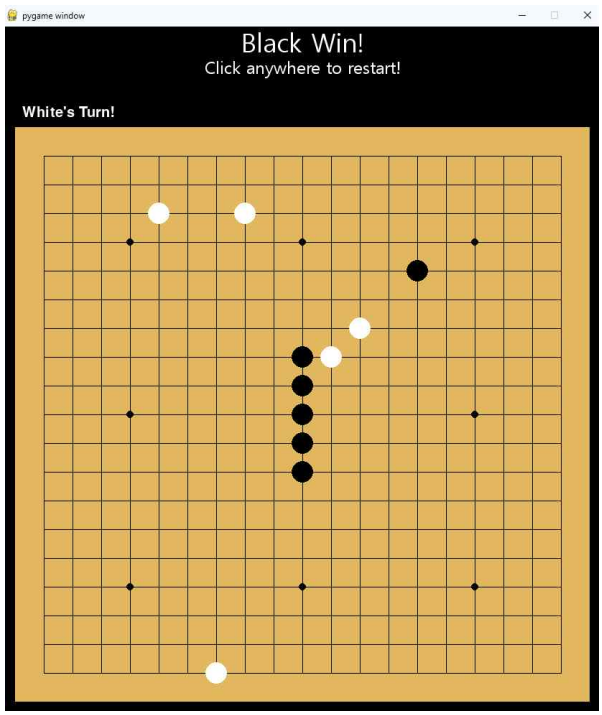
```
def get_valid_moves(state: OmokState, radius=2, limit=50):
    """
    현재 돌 기준 radius 범위 내에서 유효한 착수 위치만 반환 (최대 limit개)
    """
    board = state.game_board
    size = state.board_size
    stones = np.argwhere(board != 0)
    candidates = set()

    if stones.size == 0:
        # 아무 돌도 없으면 중앙에 두기
        return [(size // 2, size // 2)]

    for y, x in stones:
        for dy in range(-radius, radius + 1):
            for dx in range(-radius, radius + 1):
                ny, nx = y + dy, x + dx
                if 0 <= ny < size and 0 <= nx < size:
                    if board[ny][nx] == 0:
                        candidates.add((ny, nx))

    # 후보가 너무 많으면 정렬해서 일부만 사용 (탐색 속도 향상)
    candidates = list(candidates)

    # 가까운 중심 기준으로 정렬 (우선순위 높게)
    center = np.array([size // 2, size // 2])
    candidates.sort(key=lambda pos: np.linalg.norm(np.array(pos) - center))

    return candidates[:limit]
```

♥

기준 돌을 기준으로 +-radius 범위 내의 빈 칸만 수집
AI(흑)이 흩어지지 않게 한다.
최대 후보 수를 50으로 제한하여 timeout을 방지한다.



-> 결과
현재 수, 상대 수, 다음 본인 수까지 탐색하여 상대가 4목을 만들었을 때 내가 막지 않으면 진다는 것을 예측할 수 있다.



이후에 Max_depth = 2, limit = 40으로 설정한 경우 ->

-백이 흑의 4목을 막지 못했다.
탐색 깊이가 2이므로 현재 수와 다음 수까지만 본다.

≫ user ai vs ai agent의 경우 대부분의 경기에서 user agent가 승리한다.
위의 상태에서 더 개선해보려고 시도하였으나 제출한 코드가 최선인 것으로 보여졌다.

Ⅲ. 최종 코드에서 Alpha-Beta Pruning 구현 방법

def alpha_beta(state, depth, alpha, beta, maximizing_player, start_time):
-state: 현재 오목판 상태
-depth: 지금 몇 수 앞까지 보고 있는지(종료조건)
-alpha: 흑이 찾은 가장 좋은 점수
-beta: 백이 찾은 가장 나쁜 점수
-maximizing_player: 나의 턴인지, 상대의 턴인지 구분
-start_time: 타임아웃 체크용 시간

구조 요약:

```
if maximizing_player:
    max_score = -inf
    for move in possible_moves:
        new_state = 복사된 상태에 착수
        score, _ = alpha_beta(...)  # 다음 턴은 최소화
        max_score = max(max_score, score)
        alpha = max(alpha, score)
        if beta <= alpha:
            break  # 가지치기 발생!
    return max_score, best_move
```

->maximizing_player == false이면 최소값을 구하는 구조

```
if beta <= alpha:
    break  # 더 볼 필요 없음
```

-> 지금 보고 있는 가지가 더 좋은 결과를 줄 수 없다는 것이 확정
아래는 안 봐도 된다->연산량이 줄고 timeout 또한 줄어든다.