컴퓨터 인공지능학부 202312720 안수빈

1. user_agent.py의 코드 설명 및 분석 (Alpha-Beta Pruning 관련 함수 설명)

1.1) act(state: OmokState)
act(state: OmokState) 함수는 AI의 행동을 결정하는 핵심 함수로, 현재 게임 상태를 분석하여 다음 수를 반환합니다. 수를 결정하는 우선순위는 다음과 같이 구성되어 있습니다.
첫째, 본인이 4목을 완성하여 즉시 승리할 수 있는 수가 있다면 해당 수를 가장 먼저 선택합니다. 둘째, 상대가 4목을 형성해 다음 수에 승리할 가능성이 있을 경우 이를 즉시 차단합니다. 셋째, 상대가 3목을 만든 경우 당장은 큰 위협이 아니지만 이때 막지 않을 경우 오목이 될 확률이 높기 때문에 선제적으로 이를 막아 오목이되는 것을 방지합니다. 넷째, 위급한 상황이 없을 경우 minimax() 함수를 호출하여 Alpha-Beta Pruning 기반의 전략적인 수를 선택합니다. (minimax() 함수: Alpha-Beta Pruning이 적용된 트리 탐색 알고리즘) 마지막으로, 탐색 결과가 없거나 예외적인 상황에서는 후보 수 중 하나를 fallback으로 선택하여 안전하게 수를 둡니다.

1.2) minimax() (Alpha-Beta Pruning의 구현)
트리 기반 탐색은 minimax() 함수에서 구현되어 있으며, Alpha-Beta Pruning 기법이 적용되어있어 탐색 효율을 향상시킵니다. 이 함수는 현재 상태에서 가능한 모든 수를 탐색하고 그 결과를 바탕으로 가장 유리한 수를 선택합니다. 탐색 시에는 depth, alpha, beta 파라미터를 사용하여 가지치기를 수행합니다. depth는 탐색의 최대 깊이를 나타내며, 코드에서 AI는 MAX_DEPTH = 2로 설정되어 있어 2수 앞까지 예측할 수 있습니다. alpha는 최대화(maximizing) 플레이어가 현재까지 선택한 수 중 가장 높은 점수이고, beta는 최소화(minimizing) 플레이어가 현재까지 선택한 수 중 가장 낮은 점수를 의미합니다. beta <= alpha 조건이 만족될 경우, 더 이상 탐색을 진행하지 않고 해당 분기를 가지치기하여 탐색 시간을 단축합니다.
(제한 시간이 5초이기 때문에 시간 내에 처리하기 위해 MAX_DEPTH = 2로 설정하였습니다.)

2.3) get_candidate_moves() (후보의 수를 제한)
탐색 공간이 너무 넓을 경우 Alpha-Beta Pruning이라 하더라도 시간 내에 충분한 깊이를 탐색하기 어렵습니다. 이를 해결하기 위해 get_candidate_moves() 함수를 통해 후보 수를 제한하는 기법을 사용합니다. 이 함수는 전체 보드에서 모든 빈 칸을 탐색하는 대신, 이미 돌이 놓인 자리의 8방향 인접한 빈 칸들만 후보 수로 간주하여 탐색 공간을 크게 줄이는 역할을 합니다. 이렇게 사전에 필터링을 함으로써 트리 탐색의 효율을 높여 TIMEOUT = 5초로 제한되어있는 경우에서도 깊이 있는 전략적인 수를 찾을 수 있습니다.

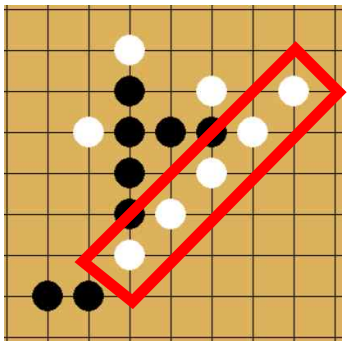
2.4) evaluate(), evaluate_direction() (평가 함수 구조 및 한계)
말단 노드의 상태를 평가하기 위해 evaluate() 함수가 사용됩니다. 이 함수는 evaluate_direction()과 함께 작동하여 각 위치의 전략적인 가치를 수치화합니다. 기준은 두 가지로 연속된 돌의 개수(count)와 양쪽이 열려 있는지 여부(open_ends)에 따라 점수를 부여합니다. 예를 들어 4목이 양쪽으로 열려 있으면 10000점, 3목이 한쪽만 열려 있을 경우 100

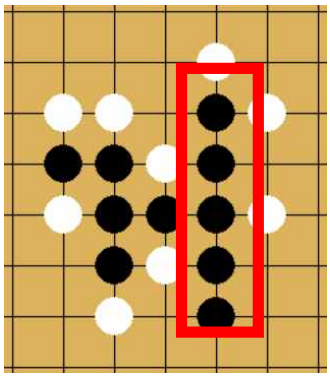
점과 같은 방식으로 가중치를 부여하여 공격과 수비의 강도를 결정하게 됩니다. 또한, 자신의 돌은 양수 점수로, 상대의 돌은 음수 점수로 계산하여 공격뿐만 아니라 수비 상황까지 고려할 수 있는 구조로 되어 있습니다.

하지만 이런 방식은 단순한 수치에 기반한 평가이기 때문에 한계가 존재하게 됩니다. 먼저 double threat(양방향 위협)과 같은 경우는 위협을 감지하지 못하며, 유도 수, 덫 수(trap) 등 맥락과 수를 읽어야하는 상위 전략은 반영되지 않았습니다. 또한 두 개 이상의 위협이 동시에 존재하는 복잡한 상황에서는 단일 점수로 정확하게 판단하기 어려운 경우가 있습니다.

2. 사용자 vs AI 대국 결과 분석 (HUMAN = True일 때)

2.1) AI가 막지 못한 경우 (AI 패) / AI가 잘 막은 경우 (AI 승)

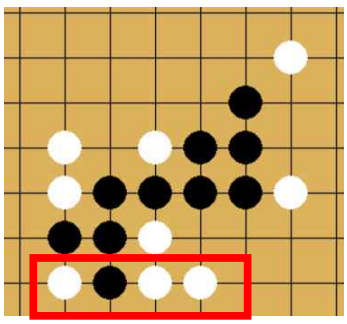




2.2) 일부러 함정 수 뒀을 때 대응 여부

2.2.1) 백-백-빈칸-백 형태의 수를 뒀을 때 (양변을 막지 않았을 때)

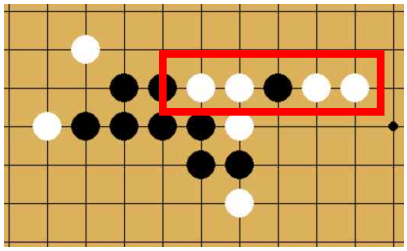
상대방(백돌)이 '백-백-빈칸-백' 형태로 놓여 있는 상황에서, 빈칸에 백돌을 두면 오목이 완성될 수 있는 위험한 패턴입니다. 실험 결과 AI는 이 형태의 위협을 인식한 후 해당 빈칸에 흑돌을 놓아 상대의 오목 완성을 방어하였습니다. 이를 통해 AI가 단순히 오목만을 만드는 것이 아니라, 즉각적인 수비 판단 능력을 이용하여 위엄한 상황을 파악 후 그에 맞는 대응을 수행하고 있다는 것을 확인할 수 있었습니다.



2.2.2) 백-백-빈칸-백-백 형태의 수를 뒀을 때

이 경우 역시 중앙의 빈칸에 백돌이 놓이면 오목이 완성되는 위협적인 상황입니다. AI는 이러한 구조에서도 위험을 감지하여 중앙의 빈칸에 흑돌을 두어 상대가 오목을 완성하는 것을 방

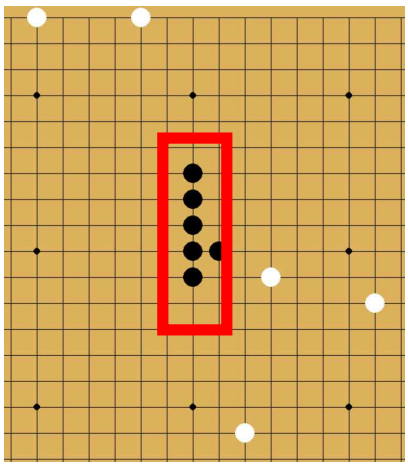
어하였습니다. 이 실험으로 AI는 단순한 4목뿐 아니라, 5목 완성 가능성을 포함한 다양한 형태의 위협 수에 대응할 수 있음을 확인할 수 있습니다



실험 번호	상황	위협	AI의 대응	결과
2.2.1	백-백-빈칸-백 (양변이 비어있음)	빈칸에 백이 들어가면 오목 완성	빈칸에 흑돌을 두고 차단	AI가 즉각적인 위협을 인식하고 방어 성공
2.2.2	백-백-빈칸-백-백	빈칸에 백이 들어가면 오목 완성	빈칸에 흑돌을 두고 차단	AI가 5목 완성 가능성을 판단하고 적절히 방어

위의 표에서 확인할 수 있듯이 상대의 오목 가능성이 있는 함정 수를 의도적으로 유도한 실험에서도 AI는 해당 위치의 위험을 감지하고 적절한 수비를 함으로써 상대의 오목 완성을 방어하였습니다. 이는 평가 함수와 find_critical_block() 함수가 정상적으로 작동하고 있으며, AI가 단순히 수를 계산하는 것뿐만 아니라, 실질적인 위협 상황에 대응할 수 있다는 것을 보여주고 있습니다.

3. 만든 AI vs 랜덤 AI와의 대결 분석 (HUMAN = False일 때)



실험 결과 만든 AI는 기본적인 수비와 공격 전략이 잘 작동함을 확인할 수 있었습니다. 본인의 오목이 완성 가능한 상황에서 이를 인식하고 마무리 수를 정확히 두는 등의 기초적인 승리

전략을 수행하였습니다. 트리 탐색을 통해 중앙 부근을 중심으로 의미 있는 위치에 돌을 배치하려는 경향도 관찰되었습니다. 하지만 반면 랜덤 AI는 특정 목표 없이 바둑판의 랜덤한 위치에 수를 두기 때문에, 수비나 공격 의도가 전혀 없으며 승리 가능성이 거의 없습니다. 따라서 실험 결과는 대부분 AI의 승리로 끝났고 여러 차례 반복해도 일관되게 유사한 결과가 나왔습니다. 물론 상대가 전략이 전혀 없는 랜덤 AI이기 때문에 대국의 의미는 미미합니다. 그러나 이번 실험은 AI가 우연히 발생하는 위협 상황에도 반응하고, 기본적인 수 평가 및 선택 로직이 정상적으로 작동하고 있는지를 확인하는 데 효과적이었습니다.

4. TIMEOUT과 MAX_DEPTH의 변화에 따른 영향

4.1) 타임아웃이 코드에서 어떤 의미인가?

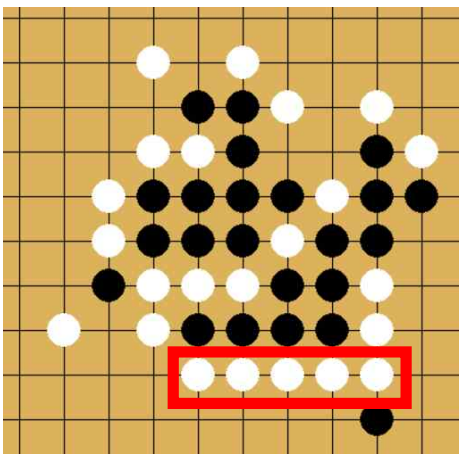
main.py 내부에 타임아웃이 설정되어 있고, 이 타임아웃은 AI가 다음 수를 계산할 때 얼마나 오랜 시간 동안 탐색을 허용할 것인지를 의미합니다. 구체적으로는 AI가 act() 함수를 통해 다음 수를 결정할 때 이 과정이 ThreadingTimeout 컨텍스트로 감싸져 있으며, TIMEOUT = 5로 설정되어 있습니다. 이는 AI가 수를 선택하는 데에 최대 5초의 시간제한이 걸려 있다는 뜻입니다. 만약 act() 함수가 5초 이내에 실행을 마치지 못할 경우에는 context_manager.state가 TIMED_OUT 상태가 되고 그때는 랜덤으로 수를 선택하게 되어 실제 AI 전략이 무효화됩니다. 따라서 이 타임아웃은 단순한 시간제한이 아니라 AI가 전략적으로 작동할 수 있는 최대 탐색 시간을 의미하며 동시에 탐색 깊이 설정(MAX_DEPTH), 후보의 수 제한, 평가 함수의 효율 등의 전체 구조 설계에 영향을 미치는 중요한 요소입니다.

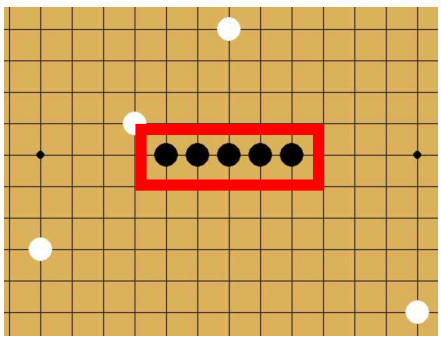
4.2) TIMEOUT과 MAX_DEPTH의 변화에 따른 영향

이 실험의 목적은 타임아웃 시간을 다르게 설정하였을 때 AI가 제한된 시간 내에 어느 정도 깊이까지 탐색할 수 있고, 또 그 탐색 깊이에 따라 성능이 어떻게 변하는지를 알아내기 위한 것입니다.

<TIMEOUT = 1초 / MAX_DEPTH = 1일 때>

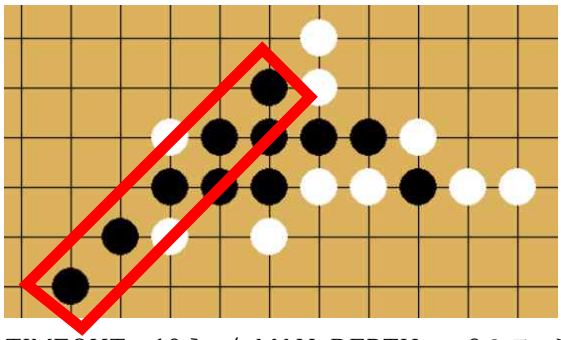
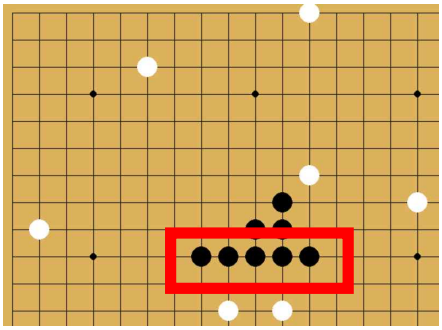
HUMAN = True (만든 AI 패) / HUMAN = False (만든 AI 승)





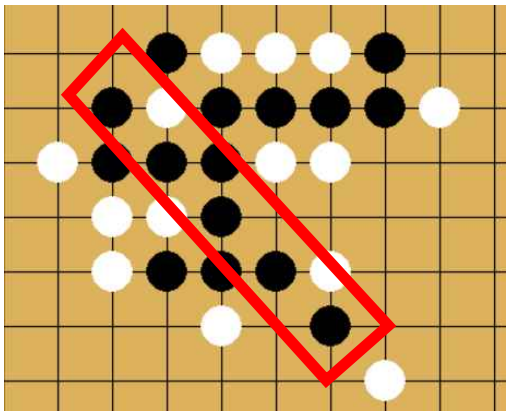
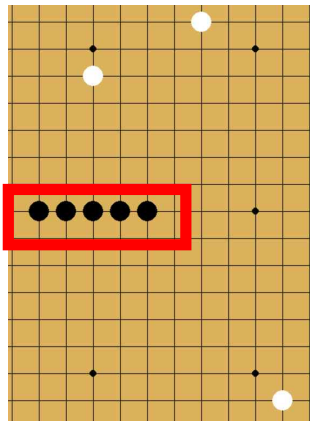
탐색 깊이가 매우 얕아 AI가 전략적 판단 없이 빠르게 수를 두는 수준에 그쳤습니다. 사용자가 직접 플레이할 경우, 수비 능력이 떨어져 TIMEOUT = 5초 / MAX_DEPTH = 2일 때 보다 쉽게 이길 수 있었습니다.

<TIMEOUT = 10초 / MAX_DEPTH = 2일 때>
HUMAN = True (만든 AI 승) / HUMAN = False (만든 AI 승)

TIMEOUT = 10초 / MAX_DEPTH = 3으로 실행시켰을 때 타임아웃이 일어나서 깊이를 2로 줄이고 실행시켰습니다. 그 결과 HUMAN = True일 때 이전보다 체감될 정도로 성능이 향상되어 AI를 이기기 어려웠습니다. 반면에 HUMAN = False일 때는 성능 면에서 이전과 다를 바가 없었습니다.

<TIMEOUT = 100초 / MAX_DEPTH = 2일 때>
HUMAN = True (만든 AI 승) / HUMAN = False (만든 AI 승)

MAX_DEPTH를 3으로 설정했을 경우에는 타임아웃 또는 에러가 발생하여서 실제 실행은 다시 깊이 2로 조정하여 진행하였습니다. 탐색 시간은 넉넉했지만, 탐색 깊이가 제한적이기 때문에 TIMEOUT = 10초일 때와 비교해도 체감 성능 차이는 크지 않았습니다. 하지만 TIMEOUT = 5초 / MAX_DEPTH = 2인 경우와 비교했을 때는 조금 더 전략적인 수를 고르는 경향이 있었으며, AI가 사용자를 상대로도 안정적으로 승리할 수 있는 수준의 성능을 유지하였습니다.

4.3) 실행시간이 증가하면서 승률도 증가하였는가?
탐색 깊이를 2로 고정한 상태에서도 TIMEOUT 값이 1초에서 10초, 100초로 늘어날수록 AI의 수 선택이 보다 안정적이고 전략적인 방향으로 변화하였습니다. 특히 HUMAN = True 모드에서 사용자와 AI가 대결했을 때 타임아웃 시간이 늘어날수록 AI의 승률이 확실히 증가하였으며 사용자가 체감하는 난이도 또한 높아졌습니다. 이는 타임아웃 시간이 늘어남에 따라 같은 깊이라도 후보 수 정렬, 평가 함수 실행, 상태 복사 등의 처리에 여유가 생기기 때문에 결과적으로 더 정밀한 수 계산과 판단이 가능해진 것으로 해석할 수 있습니다. 반면, HUMAN =

False 모드에서는 상대가 전략이 없는 랜덤 AI이기 때문에 타임아웃 시간의 변화가 게임 결과에 큰 영향을 미치지 않았습니다.