omok을 이기는 탐색 알고리즘 구현 및 결과 보고서

202312632 김예은

초기의 user_agent.py 파일은 돌의 위치를 무작위로 정하여 랜덤의 수를 두는 플레이를 하였습니다. 이는 전략이 없는 무의미한 방식으로, 오목이라는 게임의 진행과는 거리가 멀었습니다. 특별한 기능이 구현되어 있지 않기 때문에 승리하는 조건이나 패배하는 조건, 또는 흰 돌의 수에 대한 방어, 연속되게 놓아진 돌의 수 계산 등이 전혀 고려되지 않은 코드였습니다. HUMAN=True일 때, 흰 돌(내가 직접 플레이)이 연속 3개의 돌을 놓는다면 방어를 하고, 방어를 하는 것뿐만 아니라 검은 돌(ai의 플레이)이 연속 5개의 돌을 놓도록(검은 돌이 승리하도록) 시도하게끔 구현하였습니다. 돌을 두는 방식은 무작위로 두는 것이 아닌, alpha-beta pruning 탐색을 기반으로 하여서 최선의 선택을 하게 됩니다. 이 알고리즘은 AI가 앞의 수까지 탐색하며 최선의 수를 선택할 수 있도록 하는 핵심 로직입니다.

alpth-beta pruning 구현 설명 전 점수를 내는 방식을 사용한 부분을 설명하겠습니다. 현재 게임 상태에 대한 evaluate 함수를 통해 오목판 위에서의 가능한 5칸의 연속 line에 대한 분석을 기반으로 전체 점수를 산출하는 방식입니다. evaluate 함수는 오목판을 전 방향인 가로, 세로, 오른쪽 대각선, 왼쪽 대각선을 스캔을 합니다. 각각 5칸의 연속된 line을 추출한 후에 이를 evaluate_line 함수를 통해 평가 점수를 얻어냅니다. 이 점수들을 누적해서 전체 오목판 상태의 종합 점수를 산출합니다. evaluate_line 함수는 돌의 배치를 보고 다음과 같이 점수를 부여합니다.

black의 5목 완성	AI가 승리한 경우	+100,000
white의 5목 완성	대결 상대가 승리한 경우	-100,000
black의 4목	AI가 한 수만 더 두면 승리할 수 있는 경우	+15,000
white의 4목	대결 상대가 한 수만 더 두면 승리할 수 있는 경우	-50,000
black의 3목	AI가 3목을 완성한 경우	+3,000
white의 3목	대결 상대가 3목을 완성한 경우	-25,000
기타	의미가 있는 수가 아닌 경우	0

점수 부여는 몇 번의 시뮬레이션 과정을 통해 오목의 룰에 의한 의도대로 동작할 수 있는 제일 적정의 점수 값을 부여하였습니다. 예를 들어서 AI가 4목을 만드는 상황보다 대결 상대가 4목을 만들었을 때가 더 큰 영향력을 가진다는 것을 생각하여서 패배하는 것에서 먼저 벗어나는 것을 선택하게끔 수를 두게 하였습니다. 공격보다는 방어를 우선적으로 설계했음을 의미합니다. 이는 act 함수 내에서 상대의 3목이나 4목을 탐지하여 해당 위치에 돌을 놓도록 구현한 것과 일맥상통하는 부분이라고 할 수 있습니다.

다음으로 alpha-beta pruning 부분 구현입니다. 여기에서 AI(user)는 항상 1, 대결 상대는 -1로 표현하고, AI의 차례일 때는 점수를 최대화, 대결 상대의 차례일 때는 점수를 최소화하도록 설계하였습니다. 또한 탐색은 기본적으로 MAX_DEPTH =2로 설정되어 있으며, 이는 곧 'AI → 대결 상대의 차례'까지만 진행됨을 의미합니다. (깊이에 관련한 내용은 따로 밑 부분에 따로 추가하였습니다) alpha는 MAX 노드가 확보한 최소한의 값으로, 이보다 작다면 확인할

필요가 없습니다. beta는 MIN 노드가 확보한 최대한의 값으로, 이보다 크다면 확인할 필요가 없습니다. alpha와 beta는 가지치기의 조건이라고 할 수 있습니다. 우선 AI 차례에서는 점수를 최대화해야 합니다. 이때 alpha는 현재까지 알고 있는 최댓값입니다. 자식 노드의 결과가 이보다 작으면 무시합니다. 'alpha >= beta'가 되면 이후의 후보들은 모두 가지치기 됩니다. 대결 상대 차례에서 상대는 AI가 최선의 수를 두는 것을 막으려고 점수를 최소화하려고 합니다. 이 경우에는 beta 값을 유지하고, 'beta <= alpha'가 되면 가지치기 됩니다.

이에 기반하여 구현을 했기 때문에, user_agent.py와 ai_agent.py가 서로 대전하도록 했을 때 ai_agent.py는 무작위로 돌을 놓는 것에 반해 user_agent.py는 alpha-beta pruning 방식으로 동작하면서 방어를 우선적으로 합니다. 대결 상대가 놓는 연속 3개의 돌이나 4개의 돌의 위협을 탐지할 수 있고 그것을 차단하는 수를 둡니다. 그 과정에서 기회가 있을 때는 공격을 하는 수를 둡니다. 이를 종합해보면 결론적으로 user_agent.py가 더 전략적으로 탐색을 하기 때문에 win에 더 우세한, 고도화된 AI라고 볼 수 있습니다. ai_agent.py는 특별한 전략이 없이 동작하기 때문에 당연히 수 싸움에서 밀릴 수 밖에 없고 이것은 곧 패배할 확률이 높은 결과로 다가오게 됩니다.

다음은 위에서 잠깐 언급했던 깊이 제한에 관련된 내용입니다. 오목판의 공간은 매우 커서 가능한 수의 가지 수가 아주 많습니다. 초반에는 수가 수십 개 이상 존재할 것이고, 그 하위 노드까지 모두 탐색을 하는 경우에는 탐색 시간이 아주 급격하게 증가하게 될 것입니다. 우선 timeout이 발생하는 경우는 AI가 제한된 시간인 5초보다 더 오랜 시간 동안 수를 계산하느라, 주어진 제한 시간을 초과했을 때 발생합니다. 돌을 어디에 둘지 5초 이내로 결정하지 않았을 때 터미널에 timeout이 출력되면서 무작위로 수를 둡니다. 코드를 계속 개선하고 실행시키면서 많은 결과들을 확인한 결론은 돌을 둔 횟수가 많아질수록 timeout이 빈번히 발생한다는 것을 확인하였습니다. timeout이 발생하게 되면 AI가 돌을 랜덤으로 두게 되므로 제대로 된 플레이를 하지 못한다는 심각한 문제가 있었습니다.

'MAX_DEPTH = 2'가 깊이 탐색을 제한하는 코드입니다. 깊이 제한을 두지 않고 탐색한다는 것은 아무리 alpha-beta pruning을 사용한다고 하더라도 효율적이지 않다고 느꼈습니다. 이는 곧 AI가 끝까지 수를 찾기 위해서 탐색을 하다가 timeout이 발생하는 위험이 커진다는 것을 의미하기 때문에 깊이를 제한하도록 설정하였습니다. 이는 탐색 깊이가 1 증가할수록 탐색량은 폭발적으로 늘어나기 때문입니다. timeout을 발생하게 하는 제한된 시간인 5초 안에 답을 도출해내야 하므로 깊이 제한을 두는 것은 꼭 필요하다고 생각하였습니다. 이로 인해서 timeout이 발생하지 않는 선에서 가장 똑똑한 수를 둘 수 있도록 한 것입니다.

다음 수를 결정할 때 최대 탐색 깊이를 2로 제한을 해서 현재 수인 AI, 그리고 다음 수인 대결 상대의 수까지만 탐색하는 구조로, AI는 대결 상대가 반응할 수까지 고려하여 다음 수를 선택할 수 있습니다.

그리고 act 함수 내 try와 except 구조는 탐색 중에 과도한 계산으로 인해 발생할 수 있는 예외 상황을 확인합니다. 이러한 경우에 중단이 되는 것이 아니라 안전하게 처리하는 역할을 합니다. 예외가 발생하면 AI는 safe_backup 함수를 호출하여서 다음과 같은 우선순위에 의해

서 가장 안전한 수를 선택하도록 하였습니다. 백업 수를 제공하는 것은 AI가 비정상적인 timeout이 발생하더라도 항상 안전하고 의미 있는 수를 둘 수 있게 하고, 이것은 고능한 플레이를 계속 진행하도록 돕습니다.

❶ 중앙 위치가 비어 있는 경우, 그 위치에 돌을 둡니다. 이는 중앙 위치를 장악하는 것은 이기기 위한 플레이에서 전략적인 선택이라고 생각하였습니다.

❷ 중앙을 기준으로 잡고 그 주변을 확장하며 빈 칸을 탐색합니다.

❸ 이 이후에는 오목판 전체를 순차적으로 스캔하면서 비어 있는 아무 칸이라도 찾아내서 반환합니다. 최후의 선택이라고 할 수 있습니다.

이렇게 timeout이 발생했을 때 대응하는 방식을 이기기 위한 전략과 완전히 무관한 위치에 수를 두는 것보다는 중앙에서 중앙의 주변으로 확대하고, 오목판 전체로 더 확대하는 우선순위를 가짐으로써 최대한 전략적으로 접근할 수 있도록 코드를 개선하였습니다. 이로써 비정상적인 판단을 방지하고 안정성이 높아짐을 확인할 수 있습니다.

또한 탐색 노드 수를 줄이기 위한 방안으로 get_candidate_moves 함수를 구현하였습니다. 이 함수는 현재 상태에서 둘 수 있는 수 중에서 의미 있는 후보 수들만 추리는 것입니다. 비어 있는 칸 중에 주변의 8칸 중 대결 상대의 돌이나 자신의 돌이 2개 이상 있을 경우에 후보로 추가합니다. 아무런 후보가 없다면 중앙 또는 중앙 주변으로 확장시키면서 그 중 빈 칸들만 후보로 뽑도록 하였습니다. 이를 통해 실제 탐색 공간을 오목판 전체가 아니라 수를 두었을 때 의미가 있는 수가 되도록 성능을 최적화하였습니다.