Bitboard와 Alpha-Beta Pruning을 이용한 오목 프로그램 설계 및 AI의 구현

https://github.com/doobee98

1. 요약

본 과제물에서는 Python의 PyQt5 라이브러리를 활용하여 오목 GUI를 구현하고, Alpha-Beta Pruning 알고리즘을 사용하여 AI의 의사 결정 과정을 구현한다. 규칙은 과제로 제시된 룰(Custom Rule)을 기반으로 하였으며, 일반적으로 널리 사용되는 렌주룰(Renju Rule)로의 확장 역시 짧게 고려해 본다. 오목 게임 보드의 모델링은 체스 프로그래밍에 많이 사용되는 Bitboard 방법을 사용하였다.

2. 제약사항

이번 과제에서는 몇몇 제약사항이 있기에, 그것들을 이 문단에서 간단하게 정리하고 넘어간다. 추가적으로, 작성자가 임의적으로 설정한 몇몇 제약사항을 소개한다. 또한 앞으로 계속 사용할 돌 표기법을 소개한다. 자신의 돌을 O, 상대방의 돌을 X, 빈 칸을 _로 표시한다.

<과제 제약사항>

- 룰: 일반적으로 널리 통용되는 렌주룰(Renju Rule)과 달리, <33(쌍삼) 금지, 정확한 오목만 승리 가능(6목 이상은 승리하지 않는다)>을 골자로 하는 특수한 룰을 사용한다. 참고로 렌주룰은 <흑은 33, 44, 6목이상 금지, 백은 모두 허용>으로 먼저 시작하는 흑에게 페널티를 부여하는 룰이다.
- 시간제한: 10초를 기본값으로 하며, 이 값은 게임 시작시의 입력에 따라 변경 가능하다.
- 진영선택: 게임 시작시의 입력에 따라 플레이어와 AI의 진영 선택이 가능하다.
- 오목 판의 크기: 일반적으로 통용되는 15x15 크기가 아닌 19x19 바둑판을 사용한다.

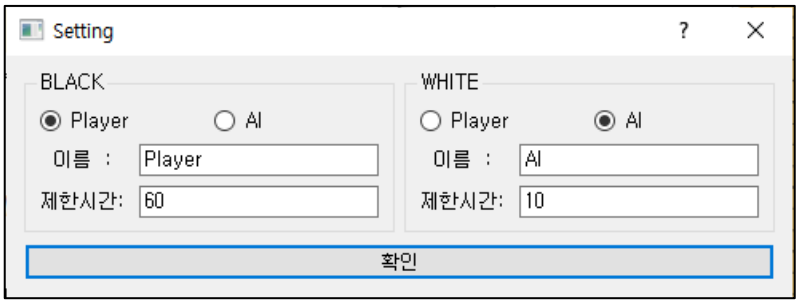
<임의 제약사항>

- 선수: 오목에서 통용되는 규칙을 사용한다(흑이 선수).
- 삼(열린3)의 정의: 오목에서 일반적으로 정의하는 "1. 상대방의 방해 없이 돌 하나를 더 두면 양쪽이 막혀 있지 않은 연결된 4(열린4)가 되는 주형, 2. 다만 4에서 하나를 더 두었을 때 승리할 수 없는 경우는 제외한다."를 3이라고 정의한다. 1번 문장의 예시로는, OOO, OO_O를 포함하며, X_OOO_X와 같은 경우는 하나를 더 두더라도 열린4가 되지 않기 때문에 3이 아니다. 2번 문장의 예시로는, XO_OO_O_OX와 같은 상태는 하나를 더 두면 열린4가 되지만, 오목을 만들 수 있는 열린4가 아니므로 3으로 생각하지 않는다.
- 무승부: 모든 칸을 다 채웠음에도 불구하고 어느 쪽도 오목을 완성하지 못한 경우 무승부가 된다.

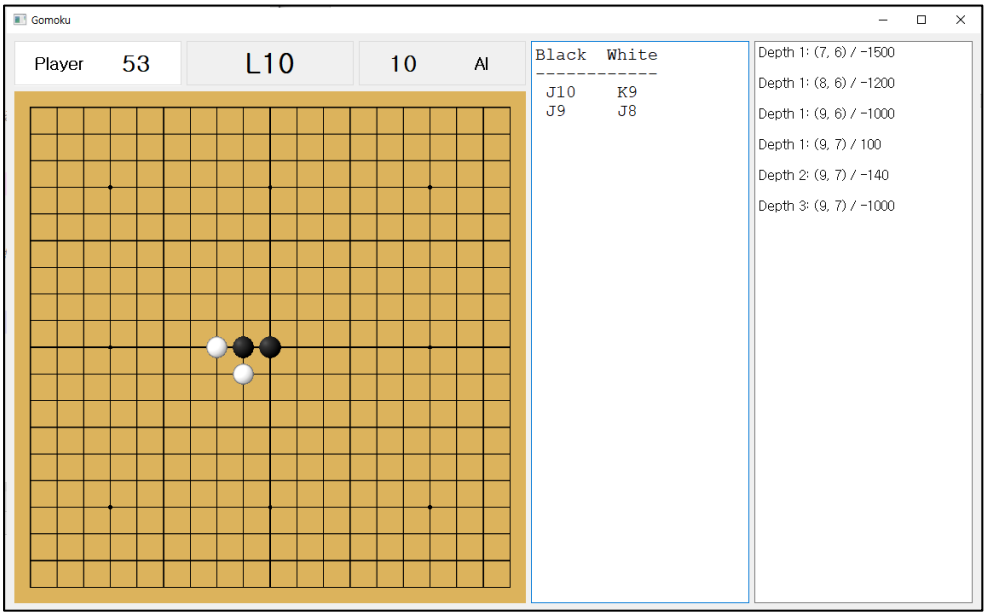
```
@pyqtSlot()
def secondOut(self) -> None:
    self.timer_count -= 1
    self.player_view[self.current_turn].setCount(self.timer_count)
    self.timer.setInterval(1000)
    if self.timer_count <= -5:  # 입력 지연 등의 처리를 위해 추가적으로 5초를 설정함
        self.timer.stop()
        self.end(self.current_turn.opponent())  # 시간 제한시 패배?
        return
```

- 시간초과시 행동: 초기에는 시간 제한을 넘길 시 바로 패배로 처리하였었는데, GUI의 입출력에 따른 약간의 시간 지연에 따른 오류 가능성 및 플레이어의 착수 편의를 위해 시간 제한이 지나더라도 추가적으로 5초의 입력 대기상태를 유지하도록 수정하였다. 상단의 시간 카운트가 -5까지 내려가는 것에서 확인할 수 있다.
- 쓰레드: AI의 행동을 결정하면서 GUI를 유지하기 위해 PyQt5의 QThread 클래스를 활용하며, 그 이상의 멀티 쓰레드는 사용하지 않는다.
- Iterative Deepening의 Reorder 제한적 사용: Transposition Table을 구현하지 않는다. 이유는 하단의 Bitboard Class를 설명할 때 서술하기로 한다.
- 의사 결정의 독립성: Search Class를 통해 의사결정을 한 후, 그 결과나 테이블을 저장하지 않는다. 즉, 각각의 수를 둘 때의 의사 결정은 독립적으로 이루어진다.

3. 오목 프로그램의 사용법, 구성 및 GUI 설계



실행파일을 실행하면 다음과 같은 세팅 화면이 나온다. 플레이어와 AI를 자유롭게 선택할 수 있으며, 이름과 제한시간을 설정할 수 있다. 제한시간은 1000 미만의 값만 입력할 수 있다.

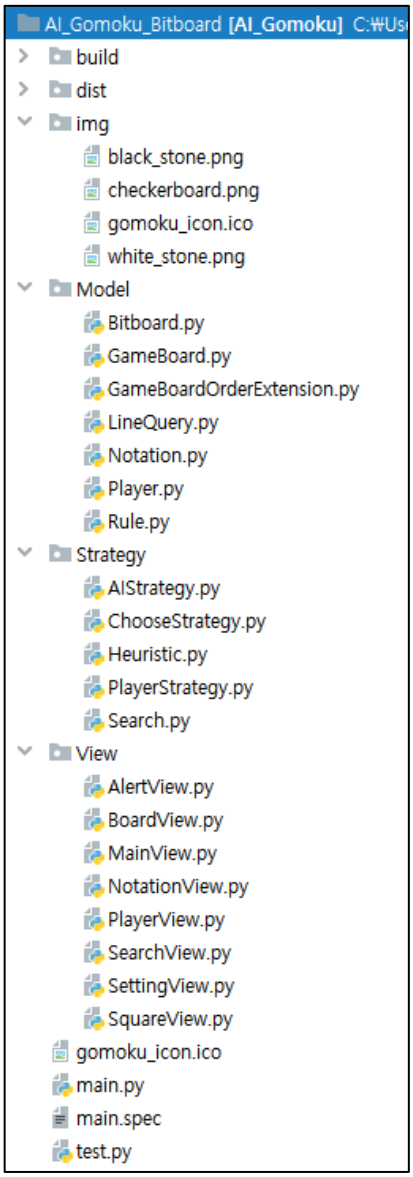


설정이 완료되면 메인 화면이 나타난다. AI는 자신의 차례가 오면 스스로 의사 결정을 하여 착수하며, 그 과정은 우측의 SearchView에서 확인할 수 있다. 플레이어는 자신의 차례에 원하는 위치에 더블 클릭을 하면 착수할 수 있으며, 금수(쌍삼)의 경우 상단의 AlertView(보드 상단 정중앙)에 쌍삼이라는 알림이 나온다. 기보 화면(NotationView)에 마우스를 올리면 지금까지 놓인 돌들의 순서를 파악할 수 있다.

제한시간이 지나면 자동으로 패배 처리가 되지만, AI의 GUI 입출력에 따른 시간 손실과 플레이어의 디버깅 편리성을 위해 추가적으로 5초가 제공된다. 해당 시간은 시간 카운트가 -5까지 내려가는 것을 통해 확인할 수 있으며, 추가로 AI는 자체적인 0.5초의 여유시간을 가진다.

콘솔 창이 아닌 GUI를 사용하기 위해 파이썬의 PyQt5 라이브러리를 활용한다. MVC 패턴을 베이스로 하여 크게 Model, View, Strategy로 분류되며, 프로그램의 로직이 복잡하지 않으므로 Main Controller의 역할은 Main View에 위임하였다. Player Controller의 역할은 ChooseStrategy 및 그 하위 클래스들이 대신한다.

View의 구조를 간단하게 설명하면, 프로그램을 실행하면 플레이어 및 AI의 설정, 이름 및 시간 제한의 설정을 위해 SettingView가 로딩되며, 필요한 정보를 입력 받은 후 main.py에서 MainView를 실행한다. 게임 보드는 BoardView에서 관리하며, SquareView에서는 흑돌 및 백돌의 렌더링과 클릭 여부를 관리한다. 프로그램의 상단에는 플레이어의 정보를 보여주는 PlayerView, 좌표 및 금수 여부를 알려주는 AlertView가 있다. 프로그램의 우측은 기보를 보여주는 NotationView, AI의 의사 결정 과정을 간략하게 보여주는 SearchView로 구성되어 있다. NotationView에 마우스 커서를 올리면 BoardView에서 현재까지 둔 돌의 순서를 표시해 주며, 해당 기능의 구현을 위해 Model에서 GameBoardOrderExtension 클래스를 추가로 정의하였다.



4. AI 구현

4-1. 룰 분석

이 단락에서는 이번 과제에서 사용하는 룰에 대해서 AI 구현에 필요한 만큼만 간단히 분석해 본다. 선수 여부에 따른 흑백의 유불리가 없기 때문에 흑백 모두 승리 플랜이 같다. 승리하기 위해서는 "1. 44 두기, 2. 43 두기, 3. 상대방의 33 금수를 유도하기, 4. 상대방의 실수"라는 네 가지 플랜이 있다. 네 가지 중 세 가지의 플랜이 두 개 이상의 줄(Multi Line)을 보아야 하기 때문에, Evaluation 함수에서 이를 반영하기 위해서는 상당히 많은 양의 연산을 해야 할 것으로 생각된다. 또한 판의 크기 역시 보통의 오목 판보다 더 크기 때문에, 연산에 있어 어려움을 겪을 것으로 보인다.

4-2. Bitboard Class

오목에서 한 칸의 상태는 흑돌, 백돌, 비어있음 세 가지이기 때문에 이진수로 표현할 수 없다. 이로 인해 이중 리스트를 사용할 시 state의 복사 및 생성에 오랜 시간이 걸리게 된다. 이 비용을 최대한 줄이기 위해 사용되는 방법 중 하나가 Bitboard이다. 아이디어를 간단히 소개하면, 흑돌과 백돌로 보드를 쪼개서 각각의 상태를 이진수로 표현하는 것이다. 흑돌의 상태, 백돌의 상태를 각각 최소 361비트의 이진수(Bitboard Class)로 표현하여 GameBoard Class에서 관리한다. 이진수 형태로 표현하기에 여러 Bit Operations, 특히 shift 연산을 사용할 수 있는 것이 가장 큰 장점이다.

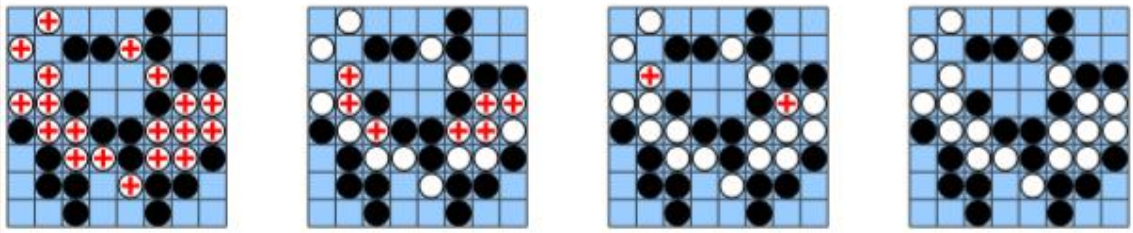
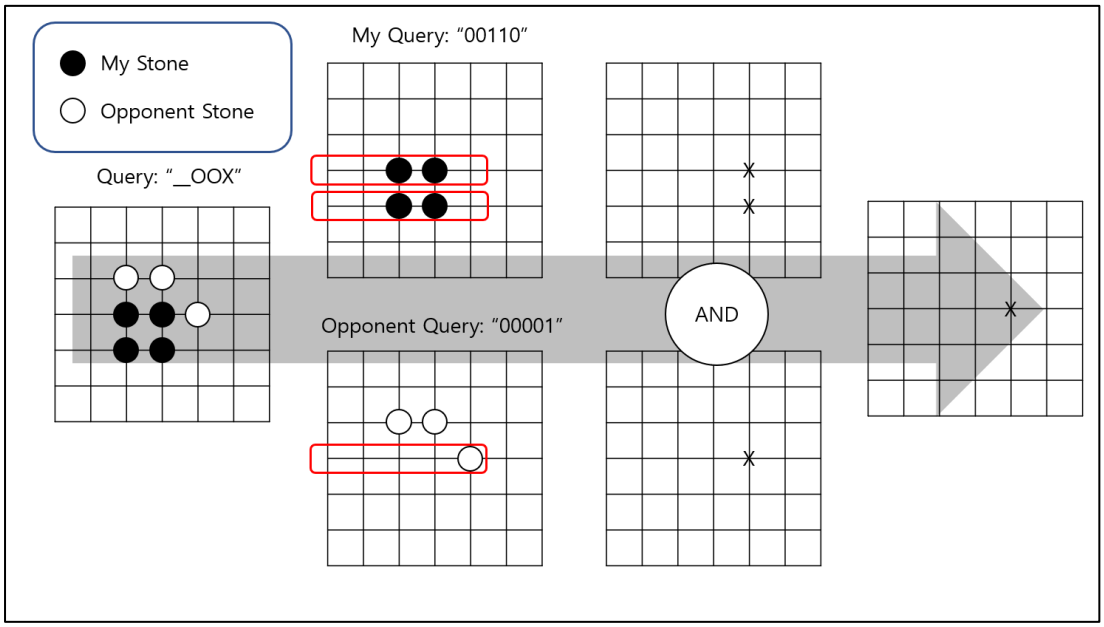


Figure 12: Erosion in the N direction detects two vertical white lines of length 3.

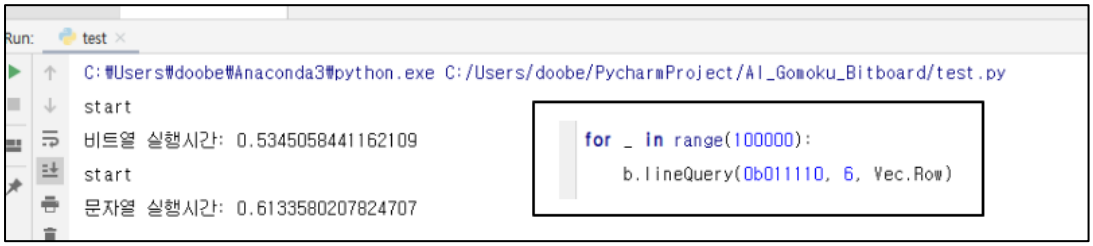
연속된 돌의 길이를 shift 연산으로 처리할 수 있다. <Bitboard Methods for Games 9p>

```
def lineQuery(self, line_query: int, line_length: int, vec: Vec) -> 'Bitboard':
    # 해당 line query가 해당 방향(vec)으로 board에 존재하면 그 위치를 반환함
    neg_board = ~self.__board
    query_iter = line_query
    bits_iter = self.__board if query_iter & 1 else neg_board
    for idx_iter in range(1, line_length):
        query_iter >>= 1
        bits_iter = (self.__board if query_iter & 1 else neg_board) & (bits_iter << Bitboard.Shift[vec])
        if bits_iter == 0:
            break
    return Bitboard(bits_iter & Bitboard.Full)
```

예를 들면, 자신의 오목을 문자열 형태로 나타내면 '_OOOOO_', 'XOOOOO_', '_OOOOOX', 'XOOOOOX' 네 가지의 경우 중 하나이다. 'XOOOOO_' 문자열을 예로 들면, 해당 쿼리를 자신의 돌과 상대방의 돌 두 종류로 나누어 비트 형태의 쿼리로 나타내면 자신의 쿼리는 '0111110', 상대방의 쿼리는 '1000000'이다. 이를 Bitboard에서 4가지 방향(Vec: Row, Col, DiagUp, DiagDown)으로 연속적인 shift와 NOT 연산을 통해 판정하면 자신이 오목을 가지고 있는지 아닌지 알 수 있다. 이렇게 shift 연산으로 쿼리 처리를 하면, 보드 전체를 한 번에 연산하면서 해당 쿼리의 존재 여부를 확인할 수 있다는 장점이 있다.



위의 그림은 자신과 상대방의 돌로 구성된 쿼리를 보드에서 Row 방향에 존재하는지 판정하는 과정을 표현한 도식이다. 각각의 색깔로 보드와 쿼리를 나눈 후 shift 연산을 진행하면, 쿼리가 존재하는 곳에는 1이 표시되고 존재하지 않는 곳에는 0이 표시된다(이때 shift 연산으로 인해 다음 행이나 열 등으로 정보가 넘어가는 것을 막기 위해 Padding, 즉 Margin이 필요하다). 각 색깔 보드의 결과값을 AND 연산하면 해당 쿼리가 존재하는 위치를 정확하게 판정할 수 있다. 위치를 정확하게 연산할 수 있다는 것은, 승리 플랜에 반드시 필요한 Multi Line 판정 구현이 쉬워진다는 것을 의미한다. 예를 들어 금수인 33을 판정하기 위해서는, 위에서 나온 결과값에서 역으로 Shift 및 OR 연산을 하면서 원래의 쿼리 상태를 복구한 후에, 복구된 3 보드끼리 AND 연산을 하면서 나온 위치가 판정하려는 금수 위치인지 확인하면 된다.



비트열을 통해 연산함으로써, 같은 쿼리의 판정을 문자열 형태보다 더 빠르게 연산할 수 있음을 위의 결과를 통해 알 수 있다. 다만 문자열 형태의 경우 라인별로 쪼개서 쿼리를 판정하고, 비트열의 경우 보드 전체를 한번에 판정하므로 Transposition Table을 구성하는 것에 있어서는 문자열 형태가 훨씬 효율적이다. 오목의 경우 체스와 달리 반복수라는 개념이 존재하지 않기 때문에(턴이 지날수록 무조건 돌의 개수가 한 개씩 증가하므로), 보드 판 전체 단위로는 중복 연산을 할 일이 거의 없다. 하지만 라인 단위로는 중복이 빈번하게 발생할 수 있기 때문에 Transposition Table을 구성하면 메모리를 효율적으로 사용해서 연산 시간을 줄일 수 있다. 작성자는 복잡한 패턴을 테스트하기 위해서 Bitboard 모델링을 선택하였다.

4-3. Search Class

이 단락에서는 AI의 의사 결정을 전반적으로 총괄하는 Search 클래스에 대해 알아본다. 사용자의 입력(게임 판 보기, 돌을 둔 순서 보기, 프로그램 즉시 종료 등)을 대기하면서 연산하기 위해 QThread 클래스를 상속하여 구현하였다.

```
# AI Decision
def run(self) -> None:
    self.start_time = time.time()
    # 만약 첫 무브라면 테이블 참조 (테이블 만들기)
    opening_search_result = self.openingTableSearch()
    if opening_search_result is not None:
        self.decision = opening_search_result
    else:
        self.alpha_beta_search(self.board)
    self.stop()
    return
```

Search Class Instance를 start하면, 세 번째 수까지는 openingTableSearch 함수를 참조하여 수를 두며, 그 외의 경우 alpha_beta_search 함수를 시행한다. 시간 제한은 Search Class의 인스턴스를 생성한 AIStrategy Class에서 측정하고 notify하며, stop 함수가 호출되면 검색이 끝났다는 신호를 방출한다.

게임 전체에서 첫 세 수 까지는(board.count() <= 2) openingTableSearch 함수를 참조한다. AI가 첫 수를 둘 경우 반드시 보드 정중앙에 두며, 두 번째 수를 둘 경우 상대방의 돌에 붙이는 랜덤한 위치에 둔다. 다만, 상대방의 보드의 가장자리(edge)에 첫 수를 두고 두 번째 수를 두어야 하는 경우에는 그냥 보드 정중앙에 둔다. 세 번째 수의 경우는 두 개의 경우로 나누어지는데. 상대방이 자신의 첫 수(보드 정중앙)에 붙여서 두지 않은 경우에는 alpha_beta_search를 하며, 붙여서 둔 경우에는 자신의 첫 수 주변 두 칸을 범위로 하는 정사각형에서 랜덤한 위치에 착수한다. 이는 게임마다 다양한 양상을 보기 위한 설정이다.

함수 alpha_beta_search에서는 min_search, max_search 함수를 이용하여 다음에 착수할 위치를 결정한다. Iterative Deepening Search 알고리즘을 활용하여 탐색 깊이를 1부터 MAX_DEPTH(설정값 5)까지 올려가며 탐색한다. Max/Min 값만 반환하면 되는 max_search, min_search 함수와 달리, alpha_beta_search 함수에서는 해당 값을 만들어 내는 다음 행동(next action)도 구해야 하기 때문에, 깊이 1에서의 첫 max_search는 해당 함수를 변형하여 alpha_beta_search 함수 내에서 계산한다.

```
# max_value를 만드는 best_action 탐색, iterative deepning
actions = Search.createActions(initial_state, 2)  # initial state는 coverage limit을 2로 둠
reorder_action_list: List[Bitboard.P] = []  # depth가 낮을때의 탐색으로 좋았던 next move를 reordering함
remove_action_list: List[Bitboard.P] = []  # 패배로 직결되는 수는 더 이상 탐색하지 않는다
```

Iterative Deepening Search를 진행하면서 Depth를 1 올릴 때, 이전 탐색결과 중 일부는 다음 탐색에 반영한다. 리스트 reorder_action_list는 탐색 도중 최대 값(max_value)를 업데이트 하는 행동(next_action_iter) 또는 최대값과 같아서 더 이상 탐색하지 않는 행동을 저장한다. 이 리스트의 값들은 다음 탐색 시에 우선적으로 반영된다. 리스트 remove_action_list는 이전 탐색 시 Heuristic.Threat_Value보다 낮은 값을 만들어 내는 행동을 저장한다. Threat_Value은 상대방의 3, 4등의 위협을 의미하며, 자세한 과정은 4-5 단락(Evaluation Function)에서 다룬다. 이 리스트의 값들은 다음 탐색 시 삭제되어 더 이상 탐색 되지 않는다.

함수 max_search와 min_search는 일반적인 alpha-beta pruning 알고리즘을 따르며, 특이점만 몇 가지 기술하려고 한다. 함수 min_search에서는 evaluation 값에 -를 붙여서 반환하고 있는데, 이는 인자로 'self.color.opponent()'을 전달함으로서 evaluation 함수가 상대방의 입장에서 계산한 점수를 반환하기 때문이다. 또한 max_search에서는 v값이 Heuristic.Win에 해당하는 값보다 크거나 같으면, min_search에서는 v값이 Heuristic.Lose에 해당하는 값보다 작거나 같으면 바로 반환함으로서 불필요한 추가 탐색을 줄인다.

클래스 함수 createActions에서는 현재 게임 보드 상태를 입력으로 받아 앞으로 탐색할 다음 행동들의 리스트를 반환한다. 추가로 받는 인자인 coverage_limit는 주변 몇 칸까지 탐색 범위에 포함시킬지를 정한다. 탐색 범위를 넓힐수록 예상치 못한 좋은 수를 잘 찾아내지만, 그만큼 탐색 시간이 크게 증가한다. 2를 넘는 coverage는 일반적으로 전투 구역에서 벗어난 큰 의미가 없는 수라고 판단하여, 본 프로그램에서는 해당 값을 2로 두었다. 이는 현재 놓여 있는 돌에서 거리가 2(대각선 거리도 1로 생각한다)만큼 떨어져 있는 위치까지 탐색함을 의미한다. Search Class에서는 Bitboard Class의 Diliation 함수를 이용하여 해당 기능을 구현하였다.

4-4. Heuristic Class

Heuristic Class는 AI의 의사 결정 과정에서 가장 중요한 evaluation 함수를 제공하는 클래스이다. 이 함수는 인자로 GameBoard state와 Current Color를 받는다. Current Color는 이제 두어야 하는 색상을 의미한다. 이 값을 입력으로 받는 이유를 닫힌 4('_OOOOX')를 예로 들어 설명해 보자. 내 차례에 내 돌이 닫힌 4를 그대로 유지하고 있다면 이는 돌 하나만 더 두면 반드시 승리하는 상황이다. 하지만 상대 차례에 내 돌이 닫힌 4를 유지하고 있는 것은 수비가 가능하다(물론 우선적으로 수비해야 한다). 반면에 양쪽 열린 4의 경우('_OOOO_') 어떤 차례이던 간에 가치가 매우 높을 것이다. 이렇듯 자신의 차례와 상대방의 차례 때에는 각 패턴들의 가치가 달라지고, 이를 반영하기 위해 패턴마다 공격 값과 방어 값을 따로 주었다.

```
Rank_Open4 = 99999
Rank_Closed4 = 95000
Rank_Open3 = 20000

Threat_Value = -Rank_Open3 + 2000

# MultiLine Value
EvalM_44 = (99999, 95000)  # 44, 열린4와 같음
EvalM_43 = (99000, 94000)  # 43, 44보다 약간 낮음
EvalM_33 = (-10000000, 10000000)  # 33, 금수

_EvaluationDict = {
    # 4개 이상
    'o000000': (100000, 100000),    # 오목
    '__0000__': (99999, 95000),     # 열린4
    'b__0000_X': (99999, 95000),    # 열린4
    'b_0_000_': (95000, 2000),      # 닫힌4
    'bX0000_o': (95000, 2000),      # 닫힌4
    'bXO_000X': (95000, 2000),      # 닫힌4
    'bX000_0o': (95000, 2000),      # 닫힌4
    'o00_00o': (95000, 2000),       # 닫힌4
    # 3개
    '__000__': (20000, 1500),       # 열린3, 완전개방
    'bX_000__': (20000, 1400),      # 열린3, 일부개방
    'bo_00_0_o': (20000, 1400),     # 열린3, 띈3
    'bX0_00_o': (3000, 600),        # 닫힌3, 띈3
    'o0_0_0o': (2000, 400),         # 닫힌3
    'bX00_0_o': (1500, 300),        # 닫힌3
    'bX000__o': (1500, 300),        # 닫힌3
    'X_000_X': (1500, 300),         # 닫힌3
```

```
# 2개
'__00__': (1500, 300),          # 열린2, 완전개방
'__0_0__': (1500, 300),         # 열린2, 완전개방
'bX_00__o': (1000, 200),        # 열린2, 일부개방
'o_0__0_o': (500, 100),         # 열린2, 띈3만 연결가능
'bX00___o': (200, 40),          # 닫힌2
```

Evaluation Dictionary를 보면, 문자열 쿼리를 Key로 하고, 해당 쿼리에 대한 Heuristic Value Tuple을 값으로 가지고 있는 것을 확인할 수 있다. 튜플의 첫 번째 값은 공격 값, 두 번째 값은 방어 값이다. 또한 Dictionary의 초기 값 위에 Multi Line 값은 따로 정의가 되어있는 것을 확인할 수 있다. Threat Value는 Search Class에서 적당한 Pruning을 하기 위해 제공되는 값이며, 상대방의 열린3 공격을 무시하고 넘길 수 없도록 -Rank_Open3을 기준으로 정의하였다. 다만 열린3보다는 닫힌4가 수비 우선순위가 더 높으므로, 닫힌4의 최대 수비 값을 빼 주었다.

이 Evaluation Dictionary는 Heuristic Class가 싱글턴 패턴으로 생성될 때 문자열 형태의 쿼리를 LineQuery로 parsing하여 인스턴스 변수인 __parsed_eval_dict에 저장된다. 이 값은 클래스 함수인 getQueryDict를 호출하여 얻을 수 있다.

다음 쪽부터는 Heuristic Value Tuple를 어떤 기준으로 책정하였는지를 서술한다. 정확한 값은 상단의 그림을 참조한다.

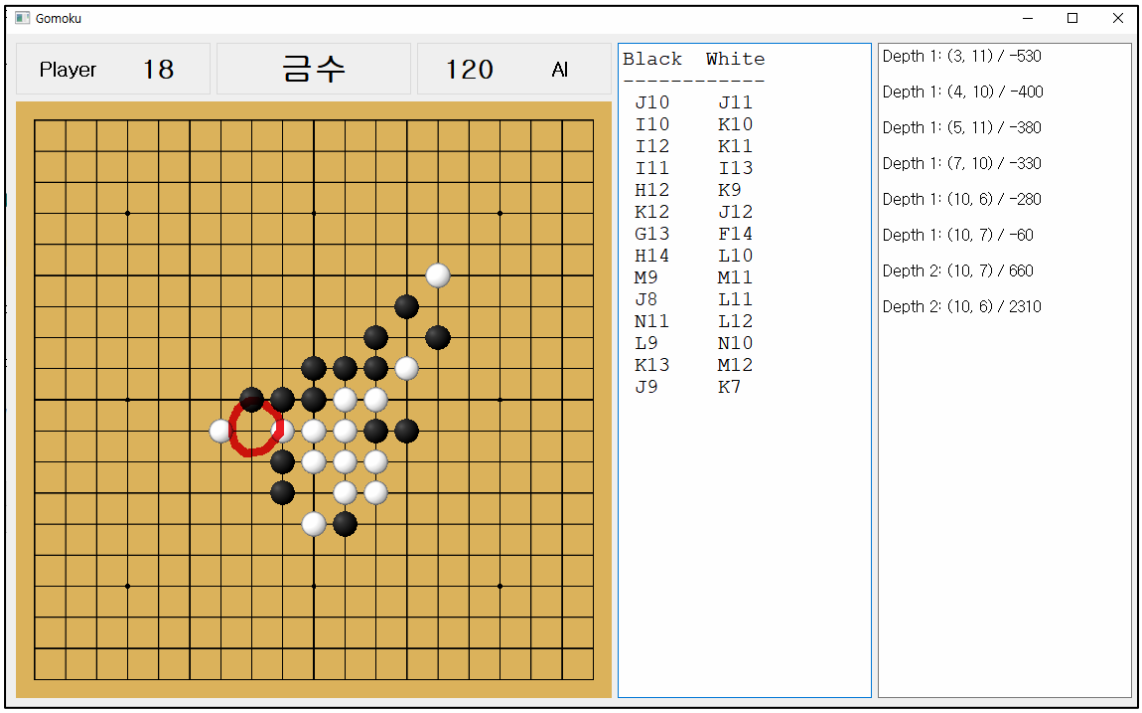
- 오목의 경우 공방에 관계없이 승리로 판정한다. 승리 및 패배 값은 10만으로 설정하였다.
- 33의 경우 자신의 돌(current_color)이라면 패배, 상대방의 돌이라면 승리한다. 금수이기 때문에 승패 값보다 더 크게 주었다.
- 열린4와 44는 공격의 경우 바로 승리할 수 있고, 방어의 경우 상대의 4가 없다면 승리한다. 상대방의 4가 있는 경우 (자신의 값) – (상대방의 값)에서 상쇄된다. 참고로 큰 의미는 없지만, 상대방의 열린4가 이미 완성된 상황에서는 다른 한 쪽을 막는 것 보다는 자신도 열린4를 만드는 것이 더 나은 선택(상대방이 실수했을 경우 바로 승리할 수 있음)이기 때문에 상쇄하는 방법을 선택하였다.
- 43 역시 승리 플랜 중 하나이지만, 내가 닫힌4를 두었을 때 상대방이 막으면서 4가 만들어진다면 수비할 수 있다. 승리까지의 시간이 조금 더 걸리는 플랜이므로 점수를 약간 더 낮게 주었다.
- 닫힌4는 공격의 경우 바로 승리할 수 있으므로 가치가 높지만, 방어의 경우에는 수비할 수는 있다. 그렇기에 공방 값의 차이가 크다. 추후 나올 열린3의 경우도 마찬가지지만, 닫힌4나 열린3과 같은 1회성 공격 착수의 방어 가치를 지나치게 높일 경우, 무의미한 공격을 계속 하는 경우가 많아진다. 따라서 닫힌4의 방어 값을 2천 정도의 공격 값과 충분히 차이가 큰 값으로 설정한 후, 나머지 값들을 이를 기준으로 설정하였다.
- 열린3은 두 수가 지나야만 승리할 수 있는 수이기 때문에, 4에 비해서는 가치가 낮은 공격이다. 열린3이 3개 있다고 하더라도 상대편이 열린4를 가지고 있다면 패배한다. 따라서 공격 값에 차이를 둘 필요가 있기에 닫힌4에 비해 약 1/5 정도의 크기인 20000으로 설정하였다. 방어값의 경우 닫힌4보다는 낮은 1500, 1400 정도의 값을 주었다. 다른 패턴에 비해 '_OOO_' 패턴의 경우는 상대방에 어떻게 막더라도 반드시 다음에 닫힌4를 만들어 낼 수 있는 수이기 때문에 약간 더 높은 점수를 주었다.
- 이후 닫힌3, 열린2, 닫힌2의 점수는 오목을 자주 두었던 작성자의 경험에 의거하여 적당하게 내림차순으로 분배해 주었다. 이 수들은 승리하는 수는 아니지만, 승리하기 위한 포석의 역할을 한다. 전투국면이 아닐 때에는 이러한 포석을 많이 만들어 두는 것이 유리한 고지를 점하는 방법 중 하나이다.

4-5. Evaluation Function

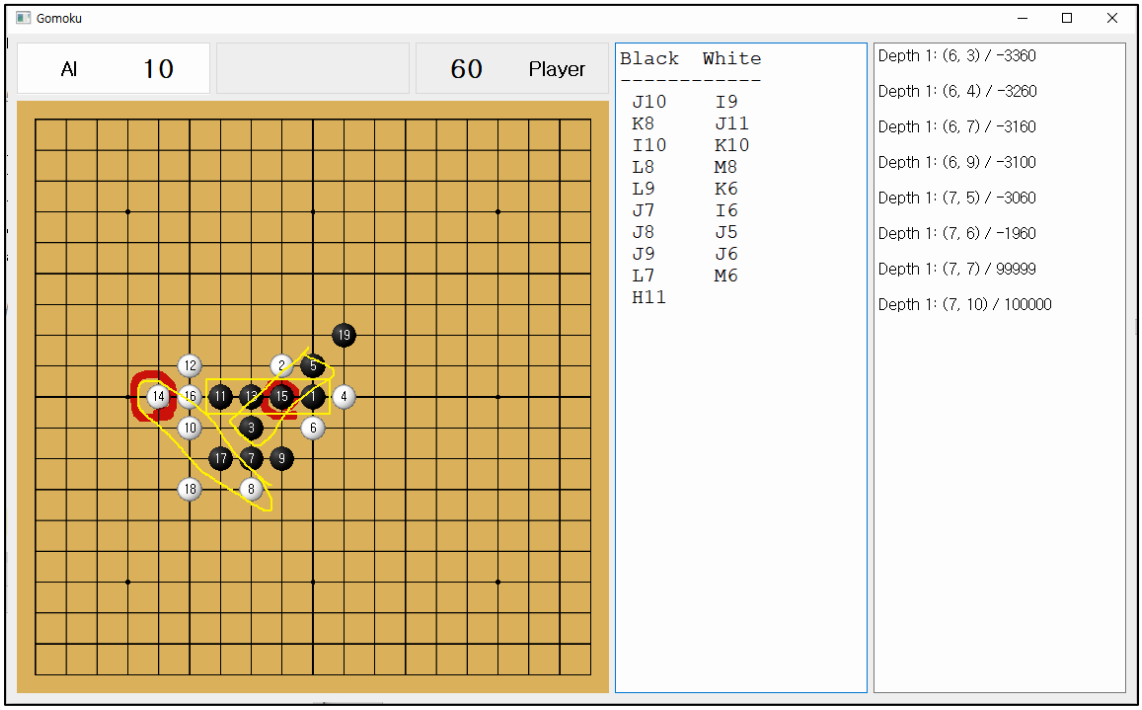
Evaluation 함수의 전체 과정은 다음과 같다.

1. getQueryDict로 문자열 형태에서 parse된 Line Query와 Evaluation Value Tuple을 받아온다. 아래의 분기에 따라 line_value를 (현재 차례의 평가 값 * 공격 값 * 패턴 등장 횟수) – (다른 차례의 평가 값 * 방어 값 * 패턴 등장 횟수)으로 계산한다.
   I. Line Query가 오목에 속하고 패턴이 인식되면, 승패를 바로 반환한다.
   II. Line Query가 열린4나 닫힌4에 속하고 패턴이 인식되면. current 혹은 incurrent pattern4 리스트에 패턴을 넣어두고 line_value 값을 바꾼다.
   III. Line Query가 열린3에 속하고 패턴이 인식되면. current 혹은 incurrent pattern3 리스트에 패턴을 넣어두고 line_value 값을 바꾼다.
   IV. 나머지 경우, line_value값을 계산하여 바꾼다.
2. Multi Line이 존재하는지 확인한다.
   A. Pattern3 리스트에 있는 패턴끼리 AND 연산을 하여 나온 위치가 최근에 둔 위치라면 승패를 바로 반환한다. 33은 금수이기 때문에 승패와 바로 연결 짓는다.
   B. Pattern4 리스트에 있는 패턴끼리 AND 연산을 하여 값이 나온다면 EvalM_44 값을 참조하여 line_value 값을 바꾼다.
   C. Pattern4 리스트에 있는 패턴과 Pattern3 리스트에 있는 패턴끼리 AND 연산을 하여 값이 나온다면 EvalM_43 값을 참조하여 line_value 값을 바꾼다.
3. line_value가 Win 값을 넘었다면 Win – 1로 변경하고 Lose 값보다 작다면 Lose + 1로 변경한 후, line_value 값을 반환한다.

상단의 과정을 통해 Heuristic Value를 계산한다. Multi Line을 계산할 때, 43과 44는 이 방법을 통해 추가적으로 연산하지 않더라도 depth +2 탐색에서 자신이 승리한다는 것을 발견할 수 있다. 하지만 depth를 2 증가시키는 것과 Evaluation 시간을 늘리는 것과 비교했을 때 전자의 시간 증가가 훨씬 크기 때문에, 조금이라도 더 정확하게 판정하기 위해서 함수 내에서 바로 판정할 수 있도록 하였다. 33의 판정은 금수인지 아닌지를 판정하는 것이기 때문에 필수적인 과정이며, 이 과정을 통해 금수 유도를 할 수 있다.



테스트 도중 오목 AI(백)가 33 금수를 인식하고 닫힌 4를 두어 승기를 잡는 모습이다.



플레이어의 33 금수 유도(14)를 무시하고 우선순위가 더 높은 자신의 43(15)를 완성시킨다.

4-6. 렌주룰로의 확장 가능성

Evaluation 함수에서 Current Color의 따라 같은 쿼리에 대해 다른 값을 부여함으로써 현재의 AI 프로그램을 손쉽게 렌주룰로 확장할 수 있다. 6목은 흑의 경우 금수이고 백의 경우 승리이기 때문에, 패턴 'OOOOOO'가 추가되어야 한다. 33과 44의 판정에서 Current Color를 참조하여 각각 다른 값을 부여하면 된다. 또한 Rule Class에서 금수를 판정하는 함수에서 흑백에 따라 44와 6목을 판정하는 과정이 필요하다.

5. 결과 정리

현재의 오목 AI는 초등학생 정도의 수준이라고 평가할 수 있다. 룰을 명확하게 이해하고 있으며 공격과 방어를 적당한 수준으로 해낸다. 하지만 상대방의 공격을 막기 위해서는 depth 2 이상의 완전 탐색을 필요로 하고, 자신의 금수 유도 회피 등을 위해서는 depth 3 이상의 완전 탐색을 필요로 하기 때문에 좋지 않아 보이는 수가 자주 나온다. 하지만 주어진 시간 안에 현재의 알고리즘으로는 depth 3의 완전 탐색이 불가능하고, 게임이 진행되면서 탐색할 범위가 점점 넓어지면서 depth 2의 완전 탐색 또한 제대로 하지 못하는 모습을 볼 수 있다. 가능한 해결책으로는 1. 파이썬이 아닌 C++등의 다른 언어 사용, 2. 전투구역 판정과 같은, 가능한 행동(next action)의 후보지를 추려내는 heuristic 알고리즘 도입, 3. 메모리의 효율적 활용을 통한 이전 상태의 저장 등이 있다. 또한 공격과 방어의 Heuristic Value 역시 약 50번간의 AI 및 플레이어 간의 테스트를 통해서 임의로 설정되었으므로, 이 역시 기계학습을 통해 더 나은 값을 찾을 것으로 기대할 수 있다.

6. 출처

1. 바둑판 이미지: https://ko.wikipedia.org/wiki/%EB%B0%94%EB%91%91%ED%8C%90
2. 흑돌 이미지: https://ko.wikipedia.org/wiki/%ED%8C%8C%EC%9D%BC:Go_b_no_bg.svg
3. 백돌 이미지: https://ko.wikipedia.org/wiki/%ED%8C%8C%EC%9D%BC:Go_w_no_bg.svg
4. Cameron Browne(2014). Bitboard Methods for Games. QUT, Brisbane, Australia