인공지능의 이해 TEAM PROJECT 2

- 목차 –

1. 카미사도의 이해

카미사도에서는 8가지 색깔로 구분된 8x8칸의 게임판을 사용한다.

카미사도는 승점제 게임와 단판 게임 2가지로 즐길 수 있는데 승점제는 단판에 비해 상대적으로 복잡하여 단판제로 만들기로 결정했다.

이 게임은 흑과 백 2명의 플레이어로 진행되는 2플레이어게임이다.

각 플레이어는 기물을 8개씩 사용하고, 각 기물에는 각기다른 8가지 색깔 중 하나를 가진다.

보드판의 양 끝 줄에 기물의 색깔에 해당하는 보드칸에 기물을 놓는다.

차례가 되면, 말 1개를 움직인다. 게임의 목적은 자신의 기물 1개를 상대방의 첫번째 끝줄의 어느 한 칸까지 이동시키는 것이다.

기물은 앞으로, 오른쪽 대각선으로, 왼쪽 대각선으로 3가지 방향 중 한 방향으로 몇 칸이든 기물에 막히지만 않는다면 전진시킬 수 있다 . 옆으로, 그리고 뒤로는 이동할 수 없다.

첫수를 두는 플레이어가 위 규칙에 따라 원하는 기물을 이동시킨다.

다음 플레이어는 첫수를 둔 플레이어가 기물을 놓은 칸의 색깔에 해당하는 기물을 이동시켜야만 한다. 이것을 번갈아가며 반복한다.

또한 기물이 블록 당하는 경우도 있다. 이동시켜야 하는 기물이 상대의 기물에 막혀있고 움직일 수 없는 경우, 지금 있는 자리로 다시 이동한 것으로 취급하여, 다시 상대 차례가 된다.

만약 블록이 연속으로 계속되어 더이상 어떤 기물도 움직일 수 없는 상태가 되면 그 상황을 유발한 플레이어가 패배하게 된다.

2. 사용된 알고리즘

easyAI 패키지의 Neagamax 알고리즘을 사용하였다. Negamax는 미니맥스(Minimax) 알고리즘의 변형 중 하나로서, 주로 턴 기반의 두 선수 게임에서 최적의 수를 찾는 데 사용되는 알고리즘이다.

Negamax는 미니맥스 알고리즘을 단순화하고 효율적으로 구현하기 위한 방법 중 하나이다. 표현 방식과 코드 구현의 간결성에 차이점이 있다. 미니맥스는 각 노드에서 최대값(Max)과 최소값(Min)을 따로 계산하여 미니맥스를 훨씬 간결하게 표현할 수 있도록 하는 특별한 케이스이다.

미니맥스(Minimax) 알고리즘은 턴 기반 게임에서 최적의 결정을 내리기 위해 사용되고 주로 보드 게임이나 카드 게임과 같은 상호작용이 있는 게임에서 적용된다. 미니맥스는 모든 가능한 게임 트리를 탐색하여 각 턴에서 최선의 수를 찾는다.

알고리즘은 두 플레이어 간의 경쟁적인 상황에서 동작하며, 하나는 "최대화(Maximize)"를, 다른 하나는 "최소화(Minimize)"를 목표한다. 각 턴에서 플레이어는 가능한 모든 수를 고려하고, 그 중에서 자신에게 가장 유리한 수를 선택한다.

미니맥스 알고리즘의 특징에는 트리 구성, 점수 할당, 반복적인 탐색, 그리고 최적의 수 선택이 있다.

트리 구성: 게임 상태를 나타내는 노드로 이루어진 게임 트리를 구성한다. 각 노드는 특정 게임 상태를 나타내며, 가지는 가능한 다음 수를 나타낸다.

점수 할당: 리프 노드에 도달하면 해당 상태의 평가 점수를 할당한다. 이는 게임이 어느 정도 어려운지 또는 어느 정도 이길 수 있는지를 나타내는 것이다.

반복적인 탐색: 게임 트리를 위에서 아래로 깊이 우선 또는 너비 우선으로 탐색하며, 각 턴에서 최대 또는 최소값을 찾아간다.

최적의 수 선택: 루트 노드에서 최종적으로 선택된 노드의 수를 플레이어에 의해 선택하고, 이를 통해 게임이 진행된다.

이러한 특징으로 최적의 수를 찾을 수 있다.

3. 게임 구현 및 코드 설명

```
class Kamisado(TwoPlayerGame):
    def __init__(self, players):
        self.players = players
        self.current_player = 1
        self.board = [[' 0'] * 8 for _ in range(8)]
        self.current_color = None
        self.setup_board()
        self.game_log = ""
        self.block_count = 0
        self.last_player = 0
```

easyAI 패키지를 사용하였다.

생성자 함수에서 플레이어와 보드를 세팅해주었다.

Borad : 8x8 사이즈의 빈 보드 생성

Current_color = 현재 움직여야 하는 말의 색

Block_count = 움직임이 막힌 횟수

Last_player = 마지막으로 움직인 플레이어

```
def setup_board(self):
        white = 7
        black = 0
        # Initialize the starting positions of towers for each player
        for i in range(8):
            self.board[0][i] = f"{white}1"  # White player tower
            white -= 1

            self.board[7][i] = f"{black}2"  # Black player tower
            black += 1
```

Setup_board()에서는 보드에 놓일 기물들을 백과 흑 그리고 각각의 색깔에 맞게 세팅해주었다.

보드의 양 끝에 일렬로 말을 배치한다.

말은 "71" 과 같은 형식이며 앞의 숫자는 7부터 0으로 말의 색을 나타내고, 뒤의 숫자는 1과 2로 플레이어를 구분한다.

```
    def get_board_color(self, i, j):
        BOARD_COLORS = [
            [7, 6, 5, 4, 3, 2, 1, 0],
            [2, 7, 4, 1, 6, 3, 0, 5],
```

```
            [1, 4, 7, 2, 5, 0, 3, 6],
            [4, 5, 6, 7, 0, 1, 2, 3],
            [3, 2, 1, 0, 7, 6, 5, 4],
            [6, 3, 0, 5, 2, 7, 4, 1],
            [5, 0, 3, 6, 1, 4, 7, 2],
            [0, 1, 2, 3, 4, 5, 6, 7]
        ]

        return BOARD_COLORS[i][j]
```

Get_board_color()에서는 말을 도착한 타일의 색을 가져온다

BOARD_COLORS에서는 보드판에 있는 칸들의 색깔을 세팅해주었다.

```
def possible_moves(self):
        moves = []
        for i in range(8):
            for j in range(8):
                if self.board[i][j][1] == str(self.current_player) and
(self.board[i][j][0] == str(self.current_color) or self.current_color ==
None):
                    moves.extend(self.valid_moves_for_piece(i, j))

        if moves == []:
            for i in range(8):
                for j in range(8):
                    if self.board[i][j][1] == str(self.current_player) and
(self.board[i][j][0] == str(self.current_color)):
                        moves.append([i, j, i, j])
        return moves
```

Possible_moves()에서는 상대가 기물을 놓은 위치에 해당하는 보드판의 색깔과 같은 자신의 기물을 이동해야 하는 경로를 찾고 moves에 추가한다. 만약 움직일 수 없다면 현재 위치를 리턴한다.

```
    def valid_moves_for_piece(self, i, j):
        moves = []
        for new_x in range(8):
            for new_y in range(8):
                if self.board[new_x][new_y] == ' 0':
                    if self.board[i][j][1] == '2' and ( (new_x < i and new_y ==
j) or ((i - new_x == abs(new_y - j) and new_x < i)) ):
                        moves.append([i, j, new_x, new_y])
                    elif self.board[i][j][1] == '1' and ( (new_x > i and new_y
== j) or ((new_x - i == abs(new_y - j) and new_x > i)) ):
                        moves.append([i, j, new_x, new_y])

        if self.current_player == 2:
            new_x = i - 1
```

```
            pivot_x = 0
            while 0 <= new_x <= 7:
                if self.board[new_x][j] != ' 0':
                    pivot_x = new_x
                elif new_x < pivot_x:
                    moves.remove([i, j, new_x, j])
                new_x -= 1

            new_x, new_y = i - 1, j - 1
            pivot_x, pivot_y = 0, 0
            while (0 <= new_x <= 7) and (0 <= new_y <= 7):
                if self.board[new_x][new_y] != ' 0':
                    pivot_x, pivot_y = new_x, new_y
                elif new_x < pivot_x and new_y < pivot_y:
                    moves.remove([i, j, new_x, new_y])
                new_x -= 1
                new_y -= 1

            new_x, new_y = i - 1, j + 1
            pivot_x, pivot_y = 0, 7
            while (0 <= new_x <= 7) and (0 <= new_y <= 7):
                if self.board[new_x][new_y] != ' 0':
                    pivot_x, pivot_y = new_x, new_y
                elif new_x < pivot_x and new_y > pivot_y:
                    moves.remove([i, j, new_x, new_y])
                new_x -= 1
                new_y += 1

        elif self.current_player == 1:
            new_x = i + 1
            pivot_x = 7
            while 0 <= new_x <= 7:
                if self.board[new_x][j] != ' 0':
                    pivot_x = new_x
                elif new_x > pivot_x:
                    moves.remove([i, j, new_x, j])
                new_x += 1

            new_x, new_y = i + 1, j - 1
            pivot_x, pivot_y = 7, 0
            while (0 <= new_x <= 7) and (0 <= new_y <= 7):
                if self.board[new_x][new_y] != ' 0':
                    pivot_x, pivot_y = new_x, new_y
                elif new_x > pivot_x and new_y < pivot_y:
                    moves.remove([i, j, new_x, new_y])
                new_x += 1
                new_y -= 1
```

```
            new_x, new_y = i + 1, j + 1
            pivot_x, pivot_y = 7, 7
            while (0 <= new_x <= 7) and (0 <= new_y <= 7):
                if self.board[new_x][new_y] != ' 0':
                    pivot_x, pivot_y = new_x, new_y
                elif new_x > pivot_x and new_y > pivot_y:
                    moves.remove([i, j, new_x, new_y])
                new_x += 1
                new_y += 1

        return moves
```

Valid_moves_for_piece()에서 기물이 이동할 수 있는 위치에 해당하는 메소드를 만들었다.

말의 위치를 받아서 보드에서 비어있고, 말의 위치를 기준으로 일직선이나 대각선 앞인 타일의 위치를 찾아서 moves에 저장한다. 그리고 움직이는 위치가 다른 말에 의해 막힌다면 moves에서 제거한다.

기물은 앞으로 오른쪽 대각선으로 왼쪽 대각선 3가지 방향 중 한 방향으로 몇 칸이든 전진시킬 수 있고 전진만 가능하도록 했다. 그리고 가려고 하는 방향에 다른 기물이 있을 경우에는 그 이후로는 못 가도록 했다. 두명의 플레이어의 방향이 반대이기 때문에 나눠서 작성했다.

```
def make_move(self, move):
        self.game_log += str(move)
        i, j, new_x, new_y = move
        if i == new_x and j == new_y:
            self.block_count += 1
            self.current_color = self.get_board_color(new_x, new_y)
        else:
            self.board[new_x][new_y] = self.board[i][j]
            self.board[i][j] = ' 0'
            self.current_color = self.get_board_color(new_x, new_y)
            self.last_player = self.current_player
            self.block_count = 0
```

Make_move()에서는 보드판에서 기물의 위치를 이동시켜주는 메소드를 만들었다.

Move를 인자로 받아서 현재 위치와 움직일 위치의 타일을 서로 바꾸고, 움직인 위치의 타일의 색을 받아와서 current_color에 저장한다. Last_player를 현재 플레이어로 설정하고 block_count를 0으로 설정한다. 만약 입력받은 move가 제자리면 움직이지않고, block_count를 1 더한다.

```
    def loss_condition(self):
        # Check if the opponent's tower reaches the opposite end
        if self.current_player == 1:
            for j in range(8):
                if self.board[0][j][1] == '2':
                    return True
```

```
        elif self.current_player == 2:
            for j in range(8):
                if self.board[7][j][1] == '1':
                    return True

        if self.block_count >= 2:
            if self.last_player == self.current_player:
                return True
        return False
```

Loss_condition()에서는 상대편의 첫번째 끝줄에 기물이 도착하는 것을 인식하는 메소드를 만들었다. 말이 보드의 반대쪽 끝에 도달하면 True를 리턴한다. 만약 block_count가 2가되면 서로 움직일 수 없는 상황이므로 먼저 교착상태를 일으킨 현재 플레이어가 True를 리턴한다.

```
    def is_over(self):
        return self.loss_condition()
```

def is_over(self):에서는 서로의 이동시켜야하는 말이 교착되어서 더 이상 이동할 수가 없거나 상대편의 첫번째 끝줄에 기물이 도착하면 게임이 종료되는 메소드를 만들었다.

```
def show(self):
        print("\n".join([" ".join([str(cell) for cell in row]) for row in
self.board]))

        color = ""
        if self.current_color == 0:
            color = "brown"
        elif self.current_color == 1:
            color = "green"
        elif self.current_color == 2:
            color = "red"
        elif self.current_color == 3:
            color = "yellow"
        elif self.current_color == 4:
            color = "pink"
        elif self.current_color == 5:
            color = "purple"
        elif self.current_color == 6:
            color = "blue"
        else:
            color = "orange"
```

```
        print(f"current color : {color}, color_num : {self.current_color}")
```

def show(self):에서는 보드판에 숫자로 적힌 색표시에 색깔명을 넣어주었다.

```
if __name__ == "__main__":
    # Initialize the game with AI and human players
    game = Kamisado([AI_Player(Negamax(3)), Human_Player])
```

2플레이어에 Negamax(3)가 쓰인 AI Player와 사람을 넣어주었다.

4. 마무리

나타난 특징: ai 플레이어가 먼저 시작할 때 항상 먼저 두는 위치 ai일때가 선공일때는 양 끝에 있는 말 중 하나를 한칸 앞으로 움직이고, ai가 후공일때는 최대한 멀리 두려고 하는 점을 발견했다. 그리고 선공은 비교적 말을 멀리 움직이지 않고 한칸씩 움직이는 반면에, 후공일 때는 모든 수를 멀리 두는 모습을 볼 수 있다.

성능: 사람과 플레이해보았을 때 어느정도의 실력을 가졌는지 게임의 규칙만 알고있는 처음 해보는 사람과 했을 때는, 확실히 승리하는 모습을 보여줬지만, 후에 기술할 필승법을 사용했을 때는 약한 모습을 보여줬다. 특히 선공일 때는 잘 대처하지 못하는 모습을 보여준다.

조건적 필승법 : ai끼리 게임을 돌려보면서 ai가 계속 특정한 수를 두는 것을 발견했다. 위 그림처럼 보드 끝에서 한칸 떨어진 위치에 내 기물 두개를 붙여서 놓을 수 있다면, 상대의 수를 강제할 수 있고, 확정적으로 승리할 수 있다.

아쉬운 점: ai가 후공일 때는 필승법 수를 두지만 선공일 때는 필승법 수를 두지 않아서 쉽게 패배한다. 실제로 ai끼리 대결할 때는 선공이 트리를 더 깊게 탐색해도 후공이 승리하는 모습을 보여준다.