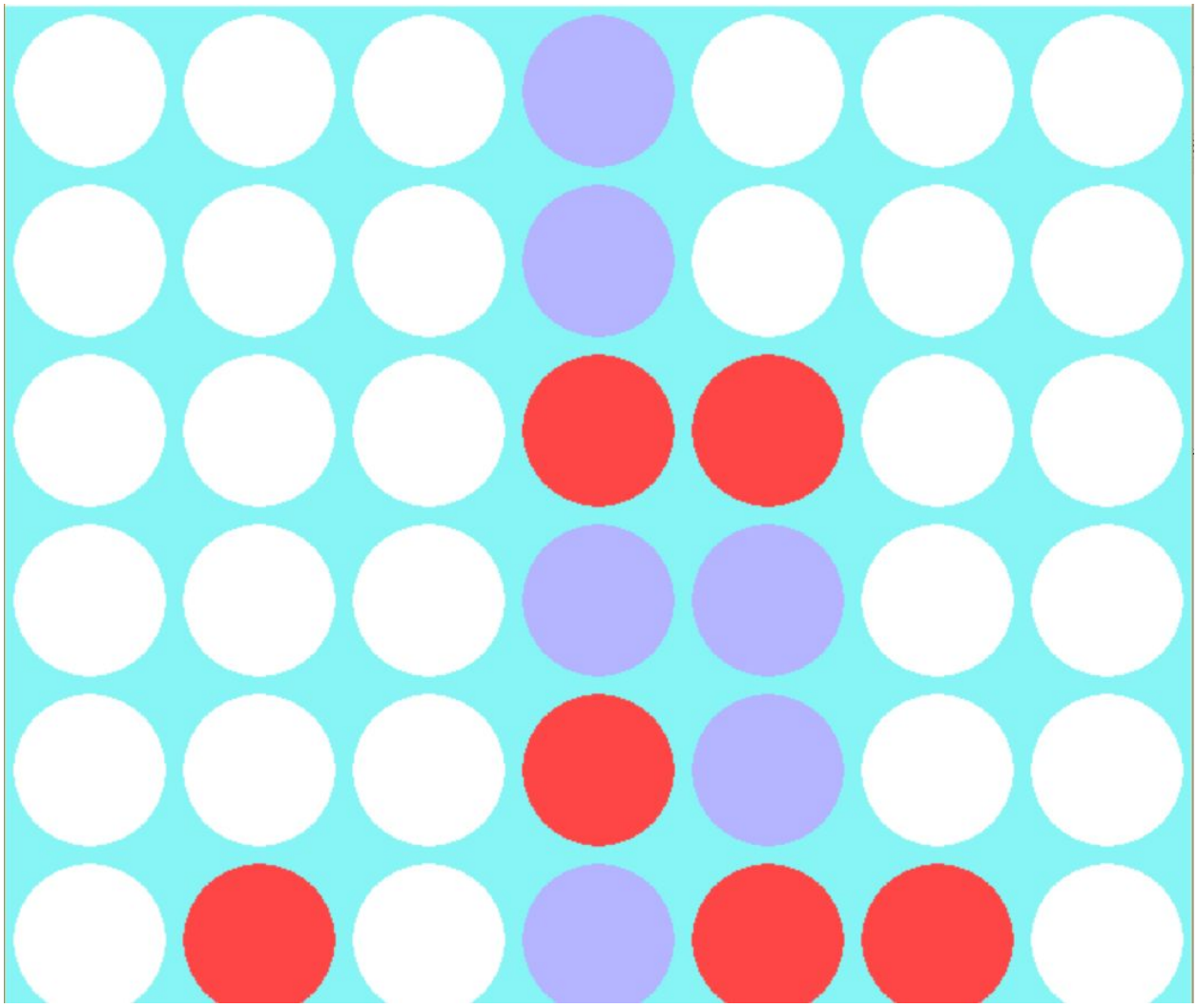


Connect Four -4조

2019.05.11

환경생태공학부 2015140521 김나연
환경생태공학부 2015140525 이수민

목차

I. 서론

Connect four 란 두 명의 player가 번갈아가면서 자신의 색깔을 가진 disc를 7×6 grid 안에 하나씩 떨어뜨리면서 진행하는 게임입니다. 먼저 자신의 색깔로 가로, 세로, 또는 대각선으로 연속된 disc를 만든 사람이 이깁니다. 처음 시작하는 사람이 먼저 가운데 (4번째 열)에 두고 시작하면 무조건 이기기 때문에 본 과제에서는 다음과 같은 제한조건을 추가하여 사람과 AI가 대전하는 형태의 Connect Four 프로그램을 작성했습니다.

A. 프로그램 명세 조건

1. 수업시간에 배운 Search 알고리즘 기반
2. 사람 vs AI 형태로 구성
3. 게임 시작 시 선공인지 후공인지 선택할 수 있어야 함
4. 게임판의 상태 출력
5. CPU 차례 시 왜 그런 결정을 내렸는지에 대한 이유를 설명하기
6. 제한 조건: 선공 시 첫수는 가운데 두지 못한다.
7. 제한 조건: 각 수 당 제한 연산시간은 2분이다.

B. 프로그램 구현 방법

4조는 Min Max Algorithm을 기반으로 하여 우리만의 Heuristic 함수를 적용했습니다. MCTS를 사용하지 않기 때문에 c언어에 비해 연산속도가 느리더라도 pygame과 numpy을 이용하기 위해 python 언어를 채택했습니다.

기본적인 Connect Four 프로그램 구현은 pygame을 이용한 GUI 방식의 https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py 에서 착안하여 작성했습니다.

II. 사용된 아이디어

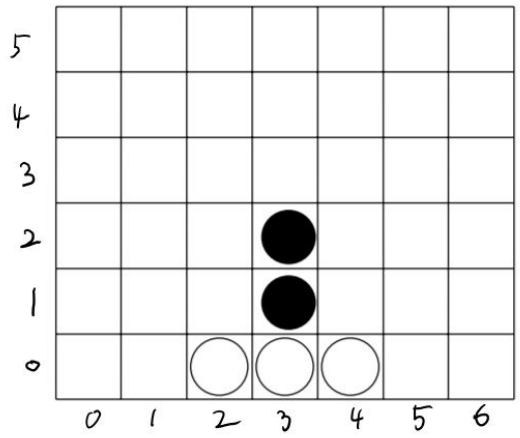
A. Heuristic

Heuristic 함수는 wikibooks.org의 Connect Four 게시물과 Victor Allis의 “A knowledge-based approach of Connect-Four-the game is solved: White wins.” 논문을 참조하여 작성했습니다.

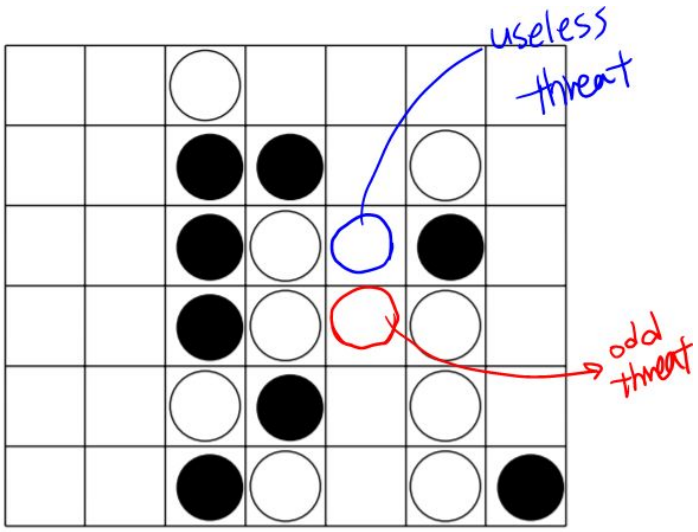
I. Threat

우선 heuristic함수에 쓰인 개념의 명칭을 명확히 할 필요가 있습니다.
Threat에 관한 이론은 대부분 Victor Allis의 “A knowledge-based approach of Connect-Four-the game is solved: White wins.” 논문에서 가져왔습니다.

- ‘**Threat**’ 이란 상대방이 놓으면 이기는 빈칸을 의미합니다.
다음 그림에서 검은 동그라미 차례라면 (0,1)과 (0,5)는 **Threat** 이 됩니다.



- Threat 중에 useless 한 threat은 버리도록 합니다. **Useless Threat**은 한 Threat의 위에 있는 Threat으로 Connect Four가 밑에서부터 채워지는 성질 때문에 고려할 필요가 없습니다.

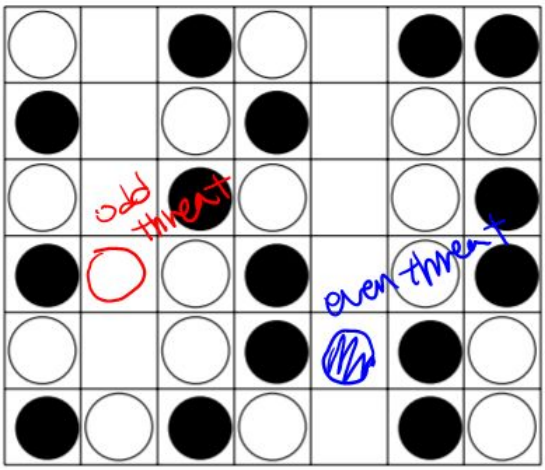


- Threat에는 **Odd threat**과 **even threat**이 있습니다. Odd 번째 row에 있으면 Odd threat, Even 번째 row에 있으면 even threat입니다.
  Useless threat, odd threat, even threat 개념은 향후 **getthreat** 함수를 구현하는 데 사용됩니다.

  이제는 선공과 후공이 각각 odd threat과 even threat을 다르게 가질 경우를 모두 정리해서 따져봅니다.

1) 선공이 Odd threat, 후공이 even threat을 가지면 선공이 이김

예를 들어 밑에 그림에서 보면 white가 선공이고 현재 black의 차례입니다. 후공은 늘 남은 빈칸 중에서 홀수번째에 말을 놓기 때문에 black은 어쩔 수 없이 자신의 even threat을 포기하고 그 밑에 놓을 수 밖에 없습니다. 따라서 선공인 white가 이깁니다.

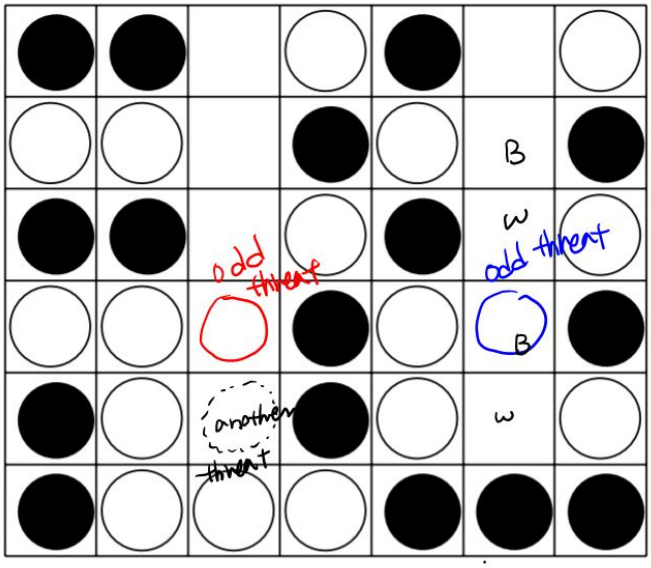


2) 선공이 even threat, 후공이 even threat 가지면 후공이 이김

후공이 우선 선공의 even threat을 무마시킨 뒤, 후공의 even threat에 말을 놓으면 이김.

3) **선공이 even threat, 후공이 odd threat을 가지면 비길 수 있다.**
4) **선공이 odd threat, 후공이 odd threat을 가지면, 후공이 이김.**

예를 들어 밑에 그림처럼 선공과 후공이 odd threat을 가지는 경우 선공은 자신의 odd threat을 포기하고 그 밑에 놓을 수 밖에 없습니다.

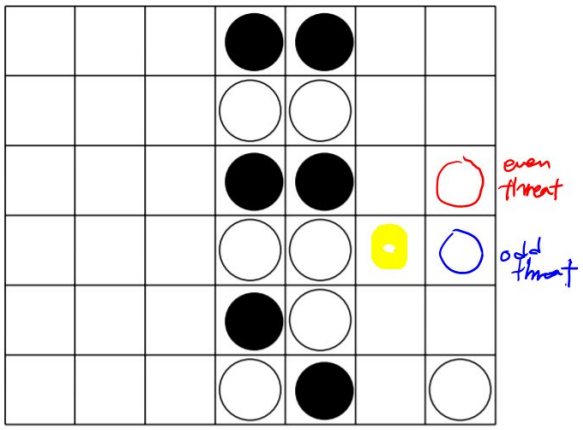


이 규칙들은 https://www.gamedev.net/forums/topic/225611-connect-4-evaluation/ 의 내용과 합쳐서 뒤에 나오는 **Odd Even Mixed** 휴리스틱 함수를 구현하는데 사용됩니다.

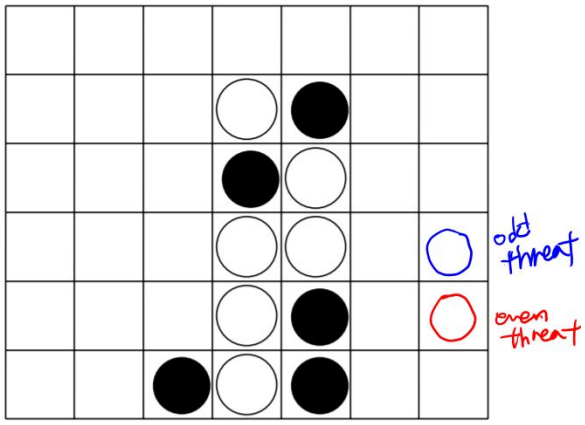
II. Double Threat

Double threat 이란 한 player의 threat 2개가 연속하게 존재하여 상대가 질 수 밖에 없는 상태를 의미합니다. 이 개념은 향후에 **getdoublethreat** 함수를 구현하는데 사용됩니다.

1) Even Threat 아래 Odd Threat



2) Odd Threat 아래 Even Threat



III. Odd Even, Mixed

- **ODD, EVEN, MIXED**
  Threat의 종류와 zugzwang의 상황을 종합적으로 다시 정리하여 ODD, EVEN, MIXED 로만 승패를 예상하는 룰을 제시한 의견을 다음 주소에서 발견했습니다.
  https://www.gamedev.net/forums/topic/225611-connect-4-evaluation/

ODD는 후공의 odd threat 갯수에서 선공의 odd threat을 뺀 개수를 의미합니다.
EVEN은 후공의 even threat 개수를 의미합니다.
MIXED 는 선공과 후공의 odd threat이 공존하는 column의 개수를 의미합니다.

그랬을 때 다음과 같은 규칙이 적용됩니다.

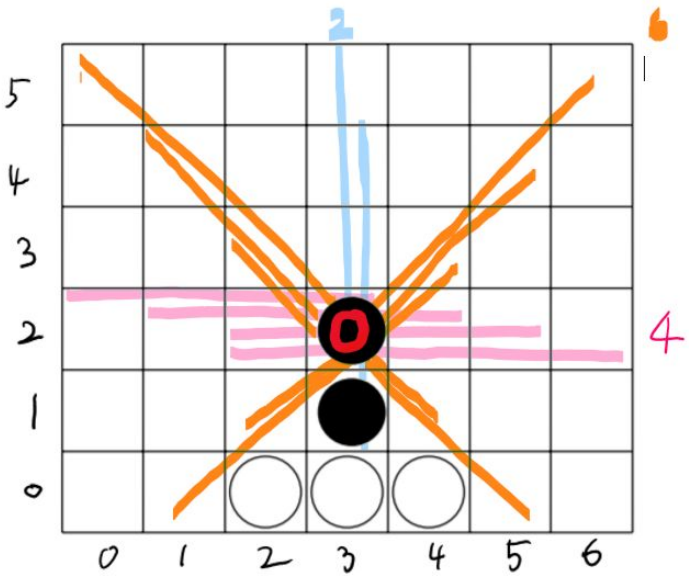
```
if (ODD<0):
      선공이 이깁니다.
if (ODD == 0):
   if (MIXED is odd)
      선공이 이깁니다.
   if(MIXED is even)
      if(MIXED ==0):
         if (EVEN==0):
               선공과 후공이 비깁니다.
         if(EVEN >0):
            후공이 이깁니다.
```

```
        if(MIXED>0):
            후공이 이깁니다.
    if (ODD == 1):
        if (MIXED is odd):
            후공이 이깁니다.
        if (MIXED is even):
            선공이 이깁니다.
    if(ODD>1):
        후공이 이깁니다.
```

이 개념은 향후 **ODDEVENMIXED** 함수를 구현하는데 사용됩니다.

IV. Line Evaluation

휴리스틱으로 Threat 개념에 너무 의존하면 게임을 시작한 뒤 아직 몇 수를 두지 않았을때 Threat의 개수가 별로 없기 때문에 유리한 위치에 잘 도달하지 못한다는 점을 발견했습니다. **초반에도 잘 두기 위해** 이 지점에 두었을 때 내게 가능한 line의 개수는 몇개인지를 판별하는 **Line Evaluation** 함수를 만들었습니다.
우선 Line이란 현재 내 말을 포함하여 가로 세로 대각선으로 가능한 연속한 4개의 칸의 개수를 의미합니다.



예를 들어 위 그림에서 (2,3)에 대한 Line Evaluation을 하면 가로로 4개, 세로로 2개, 대각선으로 6개의 line이 가능하므로 총 12개의 line이 가능하다고 evaluate합니다. 연속한 4개의 칸이 전부 내 말 또는 빈칸 일때에만 line이 성립함을 볼 수 있습니다. 이 개념은 **check_avail_line** 함수와 **line_evaluation** 함수를 구현하는데 사용됩니다.

V. Heuristic 함수들 간의 가중치 선정

Minmax search algorithm에서 최종 평가함수 값은 다음과 같이 계산합니다.

Score = score + 나의 threat 개수*100 - 상대방의 threat의 개수*100 -상대방의 double threat의 개수*100 + OddEvenMixed 판별 결과*100 + Line evaluation 함수값

이때 line evaluation 함수값은 바로 다음 첫 수에 대해서만 값을 산출한 뒤 더해주며 나머지 threat, double threat, OddEvenMixed 는 Minmax search tree의 depth가 0이 될때까지의 값(우리 조의 경우 총 다섯수 앞)을 산출하여 더해줍니다.
이런 평가함수를 선정하게 된 이유는 IV. 고찰에서 다시 설명하겠습니다.

B. Rule

1. 이기는 수가 있을때는 무조건 거기에 둡니다
2. 상대방이 둬서 이기는 수가 있을때는 무조건 거기에 둡니다
3. 둘 수 있는 경우의 수가 많은 것을 선택합니다
    - 이 rule의 경우, heuristic, 즉 상대방과 나의 현재 상태를 통해 미래를 예측하는 방법은 초반에 두어진 수가 적을 경우에는 정보가 적어서 좋은 판단을 하지 못하는 시행착오가 있었습니다. 이를 개선하기 위하여 rule을 추가하기로 하였고, 아무정보가 없을 경우에 어떤 경우가 유리한지 생각하다가 추가하게 되었습니다. 4개가 연속 될 수 있는 경우의 수가 많은 자리에 두는 것은 당연히 나중에 더 많은 기회로 돌아오게 됩니다.

C. Min Max Algorithm

수업시간에 배운 search 알고리즘 중, connect4에 가장 적절한 것은 minimax 알고리즘에 alpha beta pruning을 적용하는 것이라고 판단했습니다. 처음에는 MCTS를 시도하려고 하였으나, 연산과정이 너무 오래걸리고, 이미 solved game인 connect4에 MCTS를 적용하는 것은 비효율적인 일이라고 판단했습니다.
Min Max Algorithm 은 상대방이 유리한 경우를 minimize 시키고 내가 유리한 경우를 maximize 시키는 방식으로 search tree를 구현합니다. 이때 연산시간을 줄이기 위하여 alpha beta pruning 을 적용하면 더이상 탐색할 필요가 없는 노드를 가지치기합니다.
이와 같은 방법으로
https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py
의 코드에서 이미 구현해놓은 **minmax** 함수 형태에서 tree의 depth가 5이고 우리가 적용한 heuristic 함수 값을 넣고 최선의 수를 구했습니다.

III. 작성한 코드

* 코드 안에서의 col, row 값들은 0부터 시작하며 프린트 해줄때에만 1-7의 범위를 갖습니다.

-2015140521 김나연

```
def check_three(board, piece) :

    row = -1
    col = -1
    valid_loc = []

    #놓을 수 있는 위치 구하기
    for c in range(COLUMN_COUNT) :
        if is_valid_location(board, c) :
            valid_loc.append([get_next_open_row(board,c), c])

    #두었을때 연속하는가?
    for x in range(len(valid_loc)) :
        b_copy = board.copy()
        drop_piece(b_copy, valid_loc[x][0], valid_loc[x][1]  , piece)
        if winning_move(b_copy, piece) :
            return valid_loc[x][0], valid_loc[x][1]

    #없다면 -1 return
    return row, col
```

이기는 수가 있는 상황에 다른 수를 둘 이유가 없고, 상대방이 이기는 수가 있는 경우에 그 수를 막아야하기 위해 사용되는 함수입니다. 모든 둘 수 있는 칸에 대해 test를 해보고 연속하는 4개의 수가 있다면 return해주는 함수입니다.

```
def getthreat(board):

    AIodd = []
    AIeven = []
    PLodd = []
    PLeven = []
    AIdoublethreat = []
    PLdoublethreat = []

    #AI
```

```
    for y in range(COLUMN_COUNT) :
        for x in range(ROW_COUNT-1, 0, -1) :
            if board[x][y] == 0 :
                b_copy = board.copy()
                drop_piece(b_copy, x, y, AI_PIECE)
                if winning_move(b_copy, AI_PIECE) :
                    if x%2 == 1 : #----------------------------------------지금은 0부터 시작하니까
                        AIeven.append([x,y])
                    else :
                        AIodd.append([x,y])
                    break

    #PLAYER
    for y in range(COLUMN_COUNT) :
        for x in range(ROW_COUNT-1, 0, -1) :
            if board[x][y] == 0 :
                b_copy = board.copy()
                drop_piece(b_copy, x, y, PLAYER_PIECE)
                if winning_move(b_copy, PLAYER_PIECE) :
                    if x%2 == 1 :
                        PLeven.append([x,y])
                    else :
                        PLodd.append([x,y])
                    break

    for x in range(ROW_COUNT) :
        for y in range(COLUMN_COUNT) :
            if board[x][y] == 0 :
                if getdoublethreat(x,y,board,AI_PIECE) :
                    AIdoublethreat.append([x,y])
                if getdoublethreat(x,y,board, PLAYER_PIECE) :
                    PLdoublethreat.append([x,y])

    return AIodd, AIeven, PLodd, PLeven, AIdoublethreat, PLdoublethreat
```

Threat의 개수를 확인하는 함수입니다.
Threat를 확인하기 위해서는 임의의 위치에 수를 둘 때 이기는 경우의 수가 있는지 확인해야합니다. Threat의 특성상 column당 가장 아래에 있는 것이 유효하기 때문에 아래부터 계산해주고, 존재한다면 바로 다음 column으로 넘어갑니다.
Getdoublethreat도 계산해줘서 합해줍니다. 함수는 아래에 있습니다.

```
def getdoublethreat(x,y, board, piece) :

    if x == 1 : #오른쪽만 검사
        if y == 1 : #밑으로만 검사
            if board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece :
                return True

        elif y == 4 : #위로만 검사
            if board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y-1] == piece and
board[x+2][y-2] == piece :
                return True

        elif y == 2 or y == 3 :
            if (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece) or \
            (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece) :
                return True


    elif x == 5 : #왼쪽만 검사
        if y == 1 : #밑으로만 검사
            if board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y+1] == piece and
board[x-2][y+2] == piece :
                return True

        elif y == 4 : #위로만 검사
            if board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y-1] == piece and
board[x-2][y-2] == piece :
                return True

        elif y == 2 or y == 3 :
            if (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y+1] == piece and
board[x-2][y+2] == piece) or \
            (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y-1] == piece and
board[x-2][y-2] == piece) :
                return True

    elif x == 2 or x == 3 or x == 4 :
        if y == 1 : #밑으로만 검사
            if (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece) or \
```

```
            (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y+1] == piece and
board[x-2][y+2] == piece) :
                return True

        elif y == 4 : #위로만 검사
            if (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y-1] == piece and
board[x+2][y-2] == piece) or \
            (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y-1] == piece and
board[x-2][y-2] == piece) :
                return True

        elif y == 2 or y == 3 :
            if (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece) or \
            (board[x+1][y] == piece and board[x+2][y] == piece and board[x+1][y+1] == piece and
board[x+2][y+2] == piece) or \
            (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y+1] == piece and
board[x-2][y+2] == piece) or \
            (board[x-1][y] == piece and board[x-2][y] == piece and board[x-1][y-1] == piece and
board[x-2][y-2] == piece) :
                return True

    return False
```

임의의 장소에 수를 두었을때, 가로로 두개 그 윗줄에 대각선으로 2개의 같은 색깔 혹은 그 아랫줄에 대각선으로 2개의 같은 색깔의 수가 있다면 true를 반환해줍니다.

```
def evaluation(board, turn) :
    score = 0

    #threat값계산
    AIodd, AIeven, PLodd, PLeven, AIdoublethreat, PLdoublethreat = getthreat(board)
    oddevenmixed = 0

    #일단 threat 개수 만큼 점수 가중치
    if turn == AI :
        oddevenmixed = ODDEVENMIXED (AI, AIodd, PLodd, PLeven) *200
        score = score + len(AIodd)*100 + len(AIeven)*100 - len(PLodd)*100 - len(PLeven)*100 +
oddevenmixed - len(PLdoublethreat)*50  #+ len(AIdoublethreat)*100
    else :
        oddevenmixed = ODDEVENMIXED (PLAYER, PLodd, AIodd, AIeven)*100
        score = score - len(AIodd)*100 - len(AIeven)*100 + len(PLodd)*100 + len(PLeven)*100 +
oddevenmixed - len(AIdoublethreat)*50 #+ len(PLdoublethreat)*100
```

```
    aithreat = len(AIodd) + len(AIeven)
    plthreat = len(PLodd) + len(PLeven)
    aidoublethreat = len(AIdoublethreat)
    pldoublethreat = len(PLdoublethreat)

    return aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed, score
```

위에서 구한 휴리스틱 값들의 합을 최종 weight을 적용하여 더해준 값을 minimax함수에 return해주는 함수입니다.

```
def minimax(board, depth, alpha, beta, maximizingPlayer, line_col, firstcol): #return : col , score, aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed
    global i
    valid_locations = get_valid_locations(board)
    #3을 못두게 하는 부분
    if free_cells == 42 and depth == 5 and 3 in valid_locations:
        valid_locations.remove(3)
random.shuffle(valid_locations)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                temp = [None,100000000000000,100000000000000,100000000000000,100000000000000,100000000000000,100000000000000,100000000000000]
                return (temp, temp)
            elif winning_move(board, PLAYER_PIECE):
                temp = [None,-100000000000000,-100000000000000,-100000000000000,-100000000000000,-100000000000000,-100000000000000,-100000000000000]
                return (temp, temp)
            else: # Game is over, no more valid moves
                temp = [None, 0,0,0,0,0,0,0]
                return (temp, temp)
        else: # Depth is zero
            score, aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed = score_position(board, AI_PIECE)
            line = line_col[firstcol]
            return ([None, score + line,aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed, line],[None, 0,0,0,0,0,0,0])
```

```
    #column, value, best, secondbest,aithreat, plthreat, aidoublethreat, pldoublethreat,
oddevenmixed
    if maximizingPlayer:
        column = random.choice(valid_locations)
        value = [column,-math.inf,0,0,0,0,0,0,0,0]
        second_value = [column,-math.inf,0,0,0,0,0,0,0,0]
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
            if depth == 5 :
                firstcol = col #column이 차례대로 되니까 유효하다 첫번째수를 1로 뒀을때 쭉 가고
그다음 2로 뒀을때 쭉
            #i += 1
            #print("max : ", depth, i, "first :",firstcol,"now : ", col)
            new_value = minimax(b_copy, depth-1, alpha, beta, False, line_col, firstcol)[0][1:8]


            if new_value[0] > value[1] :
                #print("BEST!!!!!!!!!!!!!!!!!!!!",firstcol, col, new_score, value)
                second_value = value
                new_value.insert(0,col)
                value = new_value


            alpha = max(alpha, value[1])
            if alpha >= beta:
                break
        return value, second_value
    else: # Minimizing player

        column = random.choice(valid_locations)
        value = [column,math.inf,0,0,0,0,0,0,0,0]
        second_value = [column,math.inf,0,0,0,0,0,0,0,0]
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, PLAYER_PIECE)

            new_value = minimax(b_copy, depth-1, alpha, beta, True, line_col,firstcol)[0][1:8]

            if new_value[0] < value[1] :
                second_value = value
```

```
                new_value.insert(0,col)
                value = new_value

            beta = min(beta, value[1])
            if alpha >= beta:
                break

        return value, second_value
```

기존의 minimax함수는 score값만 return을 해주었으나, threat 등 evaluation에 사용된 요인들도 print를 해주어야 하기 때문에 value라는 리스트를 만들어 계속 업데이트를 해주는 방법으로 구성을 바꾸었습니다. 기본 틀은 connect4게임의 기존의 틀을 가져온 곳에서와 같은 곳인 https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py 에 있던 것에서 가져왔습니다.처음에는 value만 업데이트를 해주고, 전역변수에 best라는 리스트를 만들어 best가 생길때마다 전역변수에 저장해주는 구조를 짜는 실수를 저질렀습니다. 값이 계속 잘못나오자 minimax 알고리즘에 대해 다시 한 번 생각을 하게 되었고, best값도 score와 같이 col당으로 새로 생성되고, 값이 업데이트 되어 그 값들을 비교해주어야한다는 것을 깨달아서, 구조를 지금과 같이 고치게 되었습니다. Linecol과 first_col의 경우 저희가 적용하는 rule에서 수를 둘때 생성 될 수 있는 line의 개수를 구해서 evaluation값에 넣어서 판단을 해주어야 하기때문에 실제로 수를 두게 될 column번호를 first_col로, 그리고 minimax 함수를 돌리기 전에 미리 구해둔 line개수를 linecol리스트를 저장하여 if depth == 0 or is_terminal: 일때 불러와서 더해주도록 하였습니다.

```
    #3을 못두게 하는 부분
    if free_cells == 42 and depth == 5 and 3 in valid_locations:
        valid_locations.remove(3)
```

이 부분은 과제의 조건을 지키기 위해 첫 수를 둘때 가운데 칸에 두지 못하게 하는 부분입니다. random.shuffle(valid_locations)는 랜덤한 순서로 검사하기 위해 추가했습니다. 랜덤을 추가함으로서 성능이 올라가는 것을 확인하였습니다.
최선의 수가 왜 이것인지 그리고 다른 수를 선택하지 않은 이유를 출력해야하기 때문에 return은 value 와 second_value라는 두개의 리스트입니다. Second_value는 value가 업데이트 될때, 그 전의 value값을 저장하고 있다가, 프린트할때 차선의 수가 왜 이것인지, 왜 채택될 수 없었는지 말해주는 역할을 합니다.

(AI turn일때 실행되는 코드)

```
if turn == AI and not game_over:
        ai_row, ai_col = check_three(board, AI_PIECE)
        p_row, p_col = check_three(board, PLAYER_PIECE)

        # rule 첫번째 : 이기는 수가 있으면 무조건 거기에 두고 판을 끝낸다.
        if ai_row > -1 :
```

```
            drop_piece(board, ai_row, ai_col, AI_PIECE)
            free_cells -=1
            print("착수점 : ", ai_col+1)
            print("이유 : (", ai_row+1,",", ai_col+1,")에 두면 이기기 때문에")

        # rule 두번째 : 이기는 수가 없고, 다음판에 막을 수 있는 지는 수가 있으면 무조건 거기에
둔다.
        # 여러개가 있으면 어떡하지? -> 지는거지 뭐
        elif p_row> -1 :
            drop_piece(board, p_row, p_col, AI_PIECE)
            free_cells -=1
            print("착수점 : ", p_col+1)
            print("이유 : (", p_row+1,",", p_col+1,")에 상대방이 두면 이기기 때문에 ai가 여기
둬야한다")

        #세번째 : 앞의 두 경우가 아니면 나에게 가장 유리한 수를 둔다.
        else :
            line_col = [0,0,0,0,0,0,0]
            valid_locations = get_valid_locations(board)
            for col in valid_locations:
                row = get_next_open_row(board, col)
                line_col[col] = line_evaluation(board, AI_PIECE,row,col)

            #print(line_col)
            value, second_value = minimax(board, 5, -math.inf, math.inf, True, line_col, -1)
            print("착수점 : ",value[0]+1)
            #[None, score + line,aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed,
line]
            if value[2] == 100000000000000 :
                print("이유 : search 결과 이기는 수가 나왔기 때문에 ", value[0]+1,"을 선택하였다.")
            #best = [aithreat, plthreat, aidoublethreat, pldoublethreat, oddevenmixed,
col,value,line_col[firstcol], firstcol]
            elif value[2] == -100000000000000 :
                print("이유 : search 결과 질 수 밖에 없다. ", value[0]+1,"을 선택하였다.")
            else :
                #print("마지막 프린트시에",second_value[7])
                print("이유 : search 결과 ", value[0]+1,"이라는 수를 둘 경우 \n ai의 threat이 ", value[2],
\
                    "개, player의 threat이 ",value[3],"개, ai의 doublethreat이 ",value[4],\
                    "개, player의 doublethreat이 ",value[5],"개, oddevenmixed함수에 의한 값이
",value[6],\
                    ", 생성 가능한 line 개수에 의한 값이", value[7],\
```

```
                ", 총",value[1],"의 heuristic 값을 가지게 되어 가장 높은 heuristic값을 가지게 되므로 ",\
                value[0]+1, "에 수를 두는 것이 최선이라고 판단된다.")
            if second_value[2] == 100000000000000 :
                print("다른 수를 두지 않은 이유 : search 결과 이기는 수가 여러개 나왔고 그 중", value[0]+1,"을 선택하였다.")
            elif second_value[2] == -100000000000000 :
                print("다른 수를 두지 않은 이유 : search 결과 지는 수여서 ", second_value[0]+1,"을 선택하지않았다.")
            else :
                print(second_value[0]+1,"이라는 수를 둘 경우 \n ai의 threat이 ", second_value[2], \
                "개, player의 threat이 ",second_value[3],"개, ai의 doublethreat이 ",second_value[4],\
                "개, player의 doublethreat이 ",second_value[5],"개, oddevenmixed함수에 의한 값이 ",second_value[6],\
                ", 생성 가능한 line 개수에 의한 값이", second_value[7],\
                ", 총",second_value[1],"의 heuristic 값을 가지게 되어 ", value[0]+1,"이라는 수를 둘 때 보다 낮은 heuristic값을 가지게 되므로 ",\
                second_value[0]+1, "에 수를 두는 것이 최선이 아니라고 판단된다.")
```

Ai의 턴일때 실행되는 부분입니다. 가장 먼저 자신이 이길 수 있는 수가 있는지, 혹시 당장 상대편이 수를 둘 때이기는 부분이 있는지 확인하고 없다면 column당 수를 두었을때 생길 수 있는 line 개수를 구해주고, minimax함수를 돌립니다. 나온 결과값을 바탕으로 채택 이유를 프린트 해주고, 값이 1000000000000000이거나 -100000000000000 일 때는 이기는 수 / 지는 수 이기 때문에 말을 바꿔서 출력해주었습니다.

-2015140525 이수민

다음은 ODDEVENMIXED 함수를 구현하기 위해 우선 ODD, EVEN, MIXED 값을 구하는 함수입니다.

```
##ODD
def ODD_NUM(First_odd, Second_odd):
    return len(Second_odd) - len(First_odd)

##EVEN
def EVEN_NUM(Second_even):
    return len(Second_even)

##MIXED
```

```
def MIXED_NUM(First_odd, Second_odd):
    sum_odd = First_odd + Second_odd
    mixed_col=[]
    for x in range(len(First_odd)):
        for y in range(len(Second_odd)):
            if First_odd[x][0]==Second_odd[y][0]:
                i = First_odd[x][0]
                mixed_col.append(i)
    return  len(set(mixed_col))
```

ODDEVENMIXED 함수는 현재 판세가 유리한 상황인지, 비기는 상황인지, 지는 상황인지 판별해주는 함수라고 볼 수 있습니다. WIN값은 1,0,-1 중 하나의 값을 리턴합니다.

```
def ODDEVENMIXED (WHO, First_odd, Second_odd, Second_even): ##처음 놓는 사람의 기준으로 WIN 값 나옴
    Odd = ODD_NUM(First_odd, Second_odd)
    Mixed = MIXED_NUM(First_odd, Second_odd)
    Even = EVEN_NUM(Second_even)
    WIN = 0 ##WIN = 1 이면 유리한 상황, WIN=0 이면 비기는 상황, WIN = -1이면 지는 상황
    if (Odd<0):
        WIN = 1
    if (Odd == 0):
        if (Mixed%2 == 1): ##mixed is odd
            WIN = 1
        if(Mixed%2 == 0):##mixed is even
            if(Mixed ==0):
                if(Even >0):
                    WIN = -1
            if(Mixed>0):
                    WIN = -1
    if (Odd == 1):
        if (Mixed%2 == 1): ##mixed is odd
            WIN = -1
        if (Mixed%2 == 0):##mixed is even
            WIN = -1
    if(Odd>1):
        WIN = -1
    if WHO == EVEN:
        WIN = -WIN
    return WIN
```

다음은 Line Evaluation을 위한 함수들을 구현한 코드입니다. (김나연 약 70% 기여)

우선 현재 window, 즉 연속한 4칸이 available line인지 체크하는 check_avail_line 함수입니다.
체크함과 동시에 각 경우에 대한 휴리스틱을 적용하여 합산한 결과를 리턴합니다.

```
def check_avail_line(window, piece): #check_window 변경
    avail_line= 0
    opp_piece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        opp_piece = AI_PIECE

    #if window.count(piece)+window.count(EMPTY) == 4:
        # avail_line = 1

    if window.count(piece) == 0 and window.count(EMPTY) == 4 :
        avail_line += 1
    elif window.count(piece) == 1 and window.count(EMPTY) == 3 :
        avail_line += 3
    elif window.count(piece) == 2 and window.count(EMPTY) == 2 :
        avail_line += 4
    elif window.count(opp_piece) >= 1  :
        avail_line -= 1


    return avail_line
```

그다음은 실제 available line의 개수를 리턴해주는 line_evaluation 함수입니다.
연속된 4칸을 window라고 하며 Window의 구현은
https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py 에서 구현된
`evaluate_window(window, piece)`에서 착안했고 `score_position(board, piece)` 코드를 변형하여
작성했습니다.

가로로 window를 확인할때는 window가 가능한 범위를 start부터 end라고 하고 그 범위 안에서
모든 가능한 window에 대해 check_avail_line 함수를 돌립니다.
세로로 window를 확인할때는 window가 가능한 범위를 bottom부터 top라고 하고 그 범위 안에서
모든 가능한 window에 대해 check_avail_line 함수를 돌립니다.
대각선으로 window를 확인할때는 window가 가능한 범위를 start_x, start_y부터 end_x,
end_y라고 하고 그 범위 안에서 모든 가능한 window에 대해 각각 check_avail_line 함수를
돌립니다.

최종적으로 현재 두고자 하는 (row, col)에서 모든 가능한 window에 대한 check_avail_line 함수 값의 총 합인 avail_line_num이 리턴됩니다.

```
def line_evaluation(board, piece, row, col): #score_position(board, piece) 변경
    avail_line_num = 0

    ## Horizontal check
    start = max(col-3,0)
    end = min(col+3, 6)
    for r in range(end - start - 2) :
        window = [board[row][start + r + i] for i in range(WINDOW_LENGTH)]
        avail_line_num += check_avail_line(window, piece)

    ##Vertical check
    bottom = max(row-3,0)
    top = min(row+3,5)
    for r in range(top - bottom - 2) :
        window = [board[bottom + i + r][col] for i in range(WINDOW_LENGTH)]
        avail_line_num += check_avail_line(window, piece)

    ##diagonal check

    #//
    left = min(col-start, row-bottom)
    right = min(end-col, top-row)
    start_x = col - left
    start_y = row - left
    end_x = col + right
    end_y = row + right

    #print("why",col, row, right, left, right + left - 2 )
    for r in range(right + left - 2) :
        window = [board[start_y+i+r][start_x+i+r] for i in range(WINDOW_LENGTH)]
        avail_line_num += check_avail_line(window, piece)
        #print('오른쪽대각선', r, avail_line_num)


    left = min(col - start, top-row)
    right = min(end - col, row-bottom)
    start_x = col - left
    start_y = row + left
    end_x = col + right
```

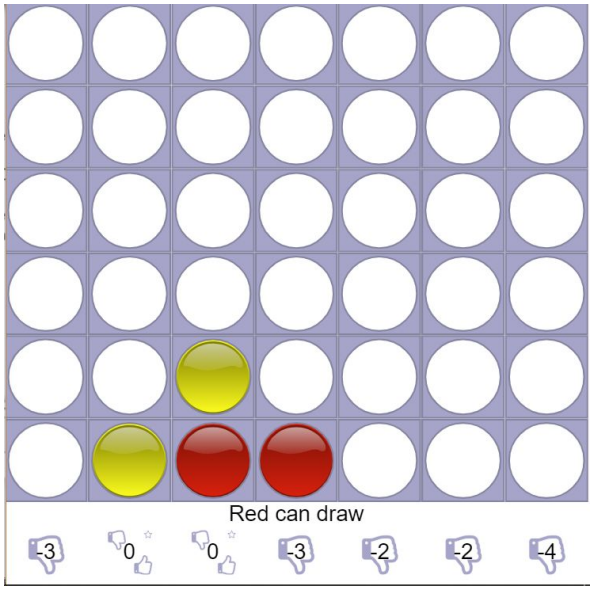
```
    end_y = row - left
    #\\

    for r in range(right + left - 2) :
        window = [board[start_y - r - i][start_x + r+i] for i in range(WINDOW_LENGTH)]
        avail_line_num += check_avail_line(window, piece)
        #print('왼쪽대각선', r, avail_line_num)

    return avail_line_num
```

IV. 고찰

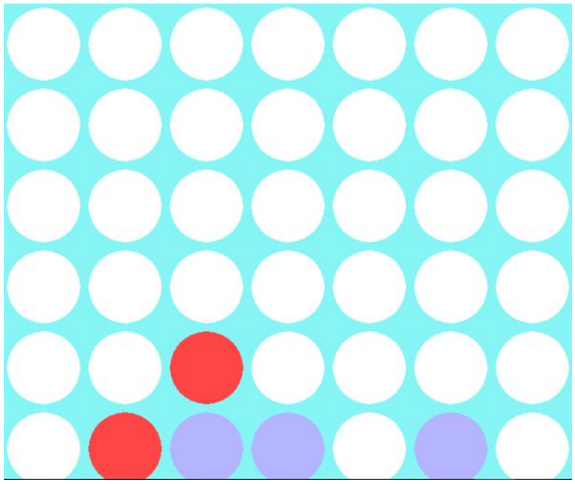
https://connect4.gamesolver.org/ 와 여러 번의 대전을 거쳐 휴리스틱 함수 간의 가중치를 조정했습니다. 그 중에서 대표적인 3가지 케이스를 예로 들어서 현재의 가중치로 정한 이유를 설명하겠습니다.

처음 평가함수 산출 방법은 모든 휴리스틱 결과에 대해 똑같이 100의 가중치를 주고 나에게 유리하면 +, 상대방에게 유리하면 -를 해서 더해주는 방식이었습니다.

1. **Double threat의 휴리스틱을 조정**
   - len(PLdoublethreat)*100 + len(AIdoublethreat)*100 에서 - len(PLdoublethreat)*100만 해주기로 결정



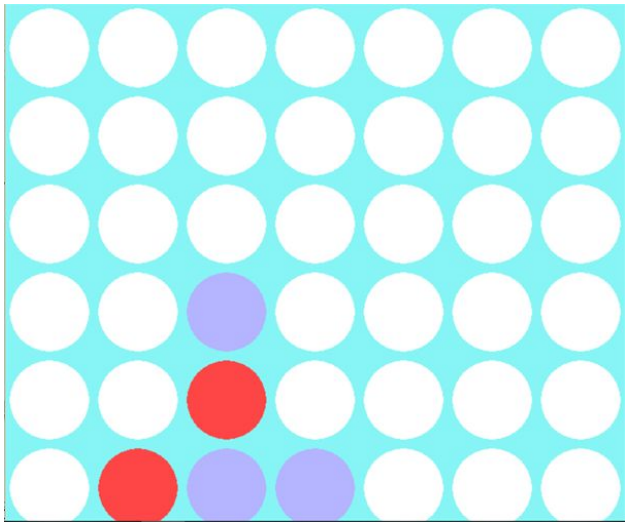
위와 같은 경우에서 connect4gamesolver 는 2번째 또는 3번째에 놓는 것이 가장 유리하다고 판단합니다.



그러나 우리의 AI는 6번째를 선택 합니다.

착수점 : **6**
이유 : search 결과 6 이라는 수를 둘 경우
ai의 threat이 0 개, player의 threat이 0 개, ai의 doublethreat이 1 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 0 , 생성 가능한 line 개수에 의한 값이 12 개, 총 62 의 heuristic 값을 가지게 되어 가장 높은 heuristic값을 가지게 되므로 6 에 수를 두는 것이 최선이라고 판단된다.
3 이라는 수를 둘 경우
ai의 threat이 0 개, player의 threat이 0 개, ai의 doublethreat이 0 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 0 , 생성 가능한 line 개수에 의한 값이 16 개, 총 16 의 heuristic 값을 가지게 되어 6 이라는 수를 둘 때 보다 낮은 heuristic값을 가지게 되므로 3 에 수를 두는 것이 최선이 아니라고 판단된다.

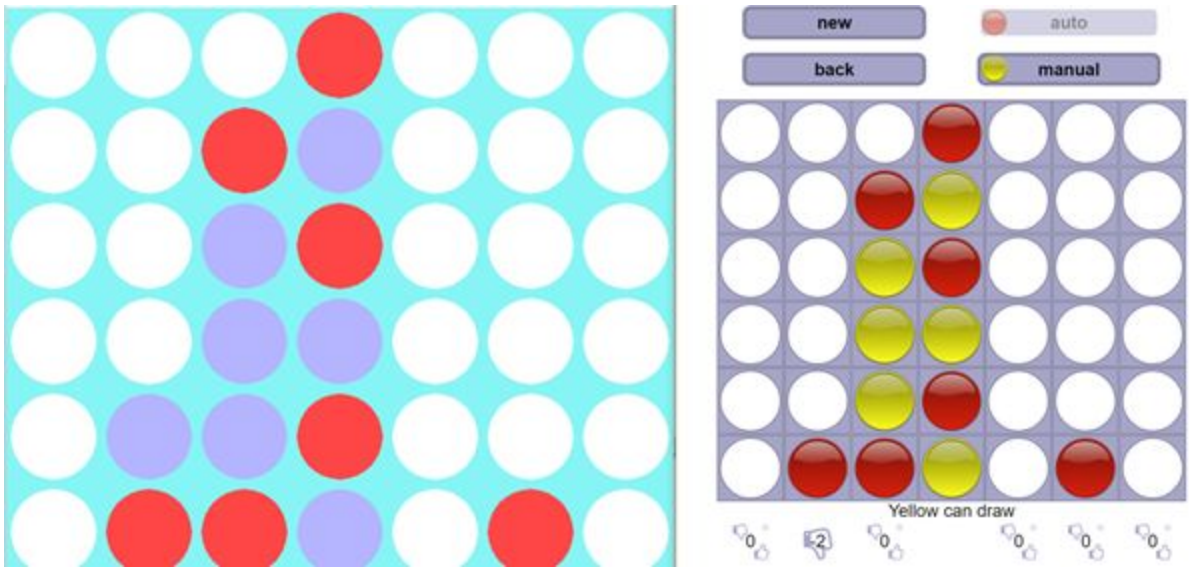
따라서 + len(AIdoublethreat)*100 부분을 삭제했습니다.



이제는 최선의 선택인 3번째에 둡니다.

앞으로 AI는 나의 doublethreat을 만드는 것보다는 상대의 doublethreat을 없애는데에 집중할 것입니다.

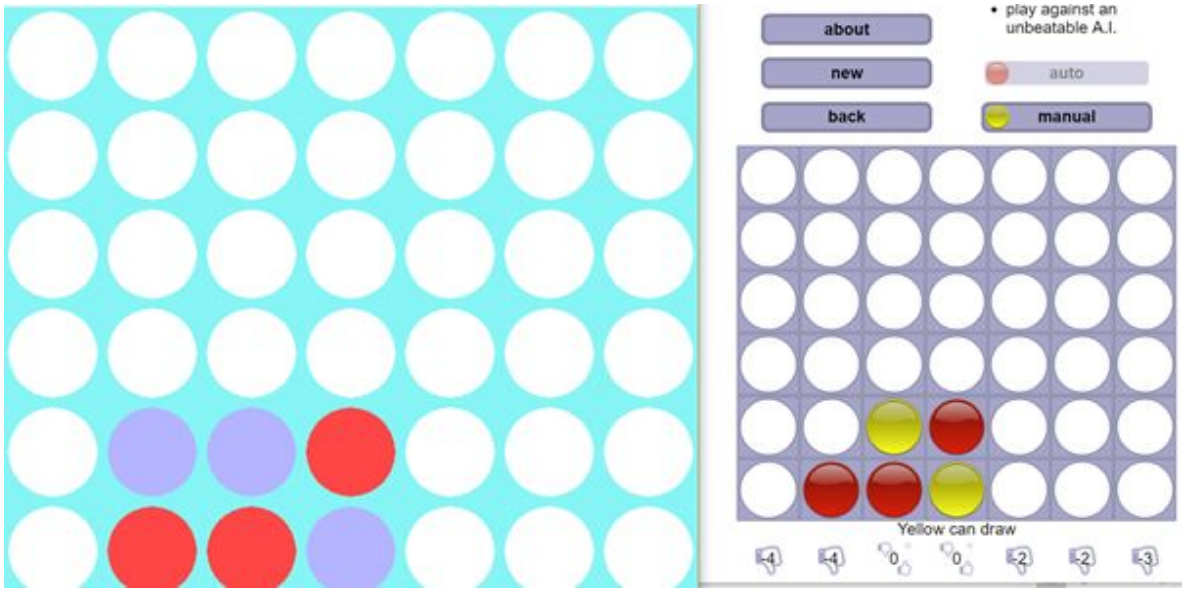
2. OddEvenMixed의 가중치는 100 으로 고정



이 상황에서 connect4gamesolver 는 1,3,5,6,7 이 유리하다고 판단합니다.
그러나 우리 AI는 2번째에 둡니다.

착수점 : 2
이유 : search 결과 2 이라는 수를 둘 경우
ai의 threat이 0 개, player의 threat이 0 개, ai의 doublethreat이 0 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 0 , 생성 가능한 line 개수에 의한 값이 7 개, 총 7 의 heuristic 값을 가지게 되어 가장 높은 heuristic값을 가지게 되므로 2 에 수를 두는 것이 최선이라고 판단된다.
1 이라는 수를 둘 경우
ai의 threat이 1 개, player의 threat이 0 개, ai의 doublethreat이 0 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 -100 , 생성 가능한 line 개수에 의한 값이 4 개, 총 4 의 heuristic 값을 가지게 되어 2 이라는 수를 둘 때 보다 낮은 heuristic값을 가지게 되므로 1 에 수를 두는 것이 최선이 아니라고 판단된다.

따라서 Oddevenmixed의 가중치를 100에서 50으로 수정합니다.



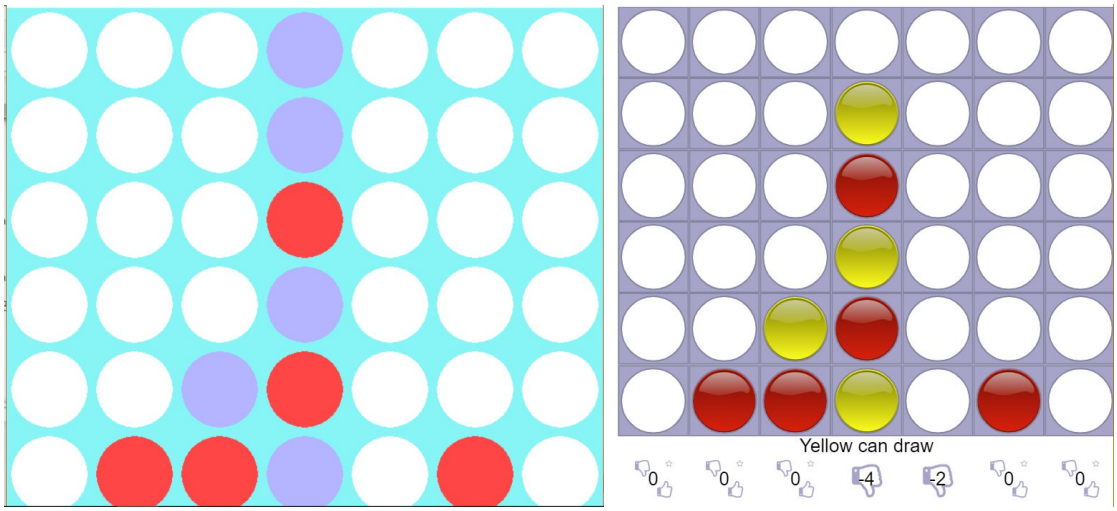
그랬더니 아예 초반부터 더 안 좋은 수를 두는 것을 확인할 수 있습니다.
따라서 Oddevenmixed의 가중치는 100 이상이어야 합니다.
Oddevenmixed의 가중치를 200으로 올리고 play해도 수의 변화는 없습니다.
따라서 가중치는 100으로 고정합니다.

3. Line Evaluation의 값 조정

처음에 Line Evaluation 값을 임의로 다음과 같이 check_avail_line 함수에서 부여했습니다.

연속된 4칸이 모두 빈 칸일때 : +1
연속된 4칸 중 3칸은 빈칸, 1칸은 내 piece일때: +2
연속된 4칸 중 2칸은 빈칸, 2칸은 내 piece일때: +3
연속된 4칸 중 상대방의 piece가 존재할때: -1

모든 가능한 연속된 4칸에 대해 각각 값을 check_avail_line 함수로 계산한 뒤 합산을 한 값이 line evaluation 값이 됩니다.



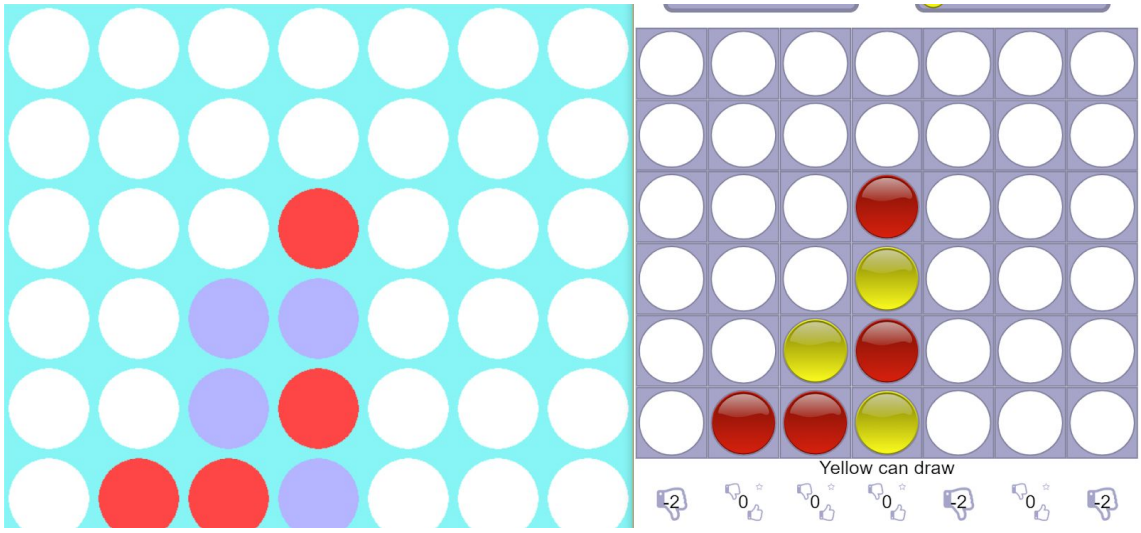
다음 경우에서 connect4gamesolver 는 4번째와 5번째가 불리하다고 하지만 우리의 AI는 4번째를 선택합니다.

착수점 : **4**
이유 : search 결과 4 이라는 수를 둘 경우
ai의 threat이 0 개, player의 threat이 0 개, ai의 doublethreat이 0 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 0 , 생성 가능한 line 개수에 의한 값이 5 개, 층 5 의 heuristic 값을 가지게 되어 가장 높은 heuristic값을 가지게 되므로 4 에 수를 두는 것이 최선이라고 판단된다.
3 이라는 수를 둘 경우
ai의 threat이 0 개, player의 threat이 0 개, ai의 doublethreat이 0 개, player의 doublethreat이 0 개, oddevenmixed함수에 의한 값이 0 , 생성 가능한 line 개수에 의한 값이 3 개, 층 3 의 heuristic 값을 가지게 되어 4 이라는 수를 둘 때 보다 낮은 heuristic값을 가지게 되므로 3 에 수를 두는 것이 최선이 아니라고 판단된다.

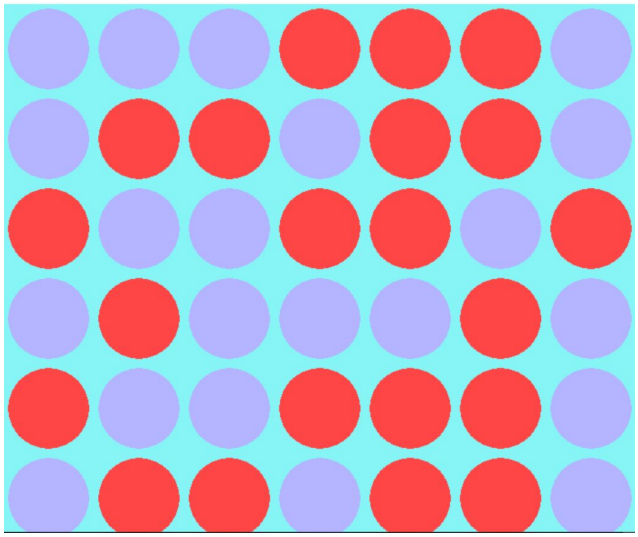
따라서 **check_avail_line 함수 내의 가중치**를 다음과 같이 **수정**했습니다.

연속된 4칸이 모두 빈 칸일때 : +1
연속된 4칸 중 3칸은 빈칸, 1칸은 내 piece일때: **+3**
연속된 4칸 중 2칸은 빈칸, 2칸은 내 piece일때: **+4**
연속된 4칸 중 상대방의 piece가 존재할때: -1

그러자 다음과 같이 수를 유리하게 변경합니다.



계속 play하게 되면 결국 connect4gamesolver와 비깁니다.



V. 참고문헌

1. Allis, V. (1988). A knowledge-based approach of Connect-Four-the game is solved: White wins.
2. https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py
3. https://en.wikibooks.org/wiki/Connect_Four
4. https://www.gamedev.net/forums/topic/225611-connect-4-evaluation/
5. https://github.com/brendanator/connect4
6. https://connect4.gamesolver.org/