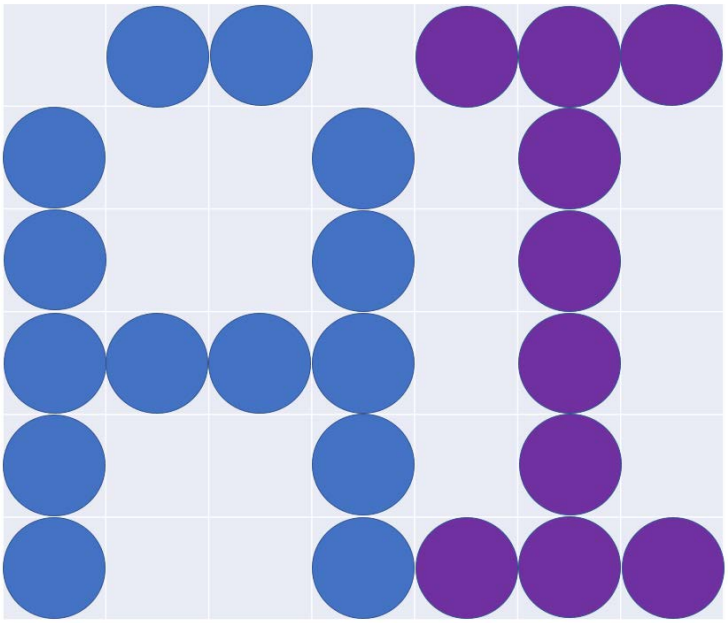
인공지능 - 02분반

제출일 : 2018.05.07

텀프로젝트 보고서

팀 이름 : 전우주

CONNECT 4 프로젝트



컴퓨터학과

2016320150 우지수

2016320152 전지연

2016320171 정예주

CONTENTS

1.알고리즘의 개념 및 배경적 지식

1) Game tree

게임 트리란 1:1 보드게임 상황에서 각 경우의 수로부터 얻는 점수들을 모두 트리 형식으로 나타내어 도식화 한 것을 말한다.

2) Minimax algorithm

게임을 할 때 나는 나에게 최적의 수, 즉 max 값을 선택할 것이고 상대는 상대에게 최적의 수, 즉 나에게는 min 값을 선택할 것이다. Minimax 알고리즘은 나와 상대방의 관점을 모두 고려하여 한 번은 주는 점수가 min 이 되는 수를, 그 다음 수는 점수가 max 가 되는 수를 선택함으로써 결과적으로 현재의 기로에서 가장 최선의 수를 선택할 수 있도록 도와주는 알고리즘이다.

3) Alpha-Beta pruning

Minimax 알고리즘은 너무나도 불필요하게 모든 경우의 수를 계산해서 자신의 수를 최종적으로 선택하게 되므로 이는 비효율적이다. 그래서 불 필요한 경우의 수는 계산하지 않도록 가지치기를 하는 방법이 alpha-beta pruning 이다.

alpha는 MAX 노드에서 lower bound이고 beta는 MIN 노드에서 upper bound 이다.

pruning을 할 때, v<=α 일때와 v>=β일 때 cut 함으로써 탐색하는 노드의 수를 효율적으로 줄일 수 있다.

4) Negamax algorithm

```
max(a,b) == -min(-a,-b);
int negaMax(int depth){
   if(depth == 0)
      return evaluate();
   int max = -oo;
   for(all moves) {
      score = -negaMax(depth - 1);
      if(score > max)
         max = score;
   }
   return max;
}
```

negamax 간단한 구현 예제

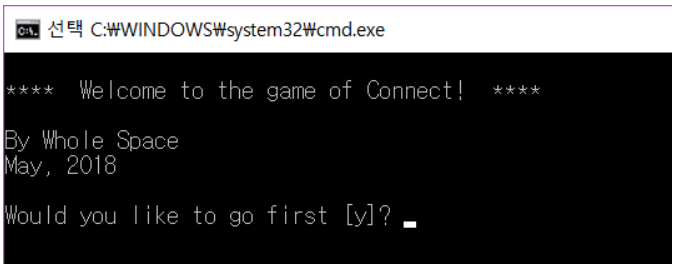
Negamax algorithm 은 Min-max search algorithm을 기반으로 한 알고리즘이다. board에 대한 player의 점수는 opponent의 board의 점수에 대한 negation이란 것이란 아이디어를 이용하여 Min-max algorithm을 발전시킨 방법이다. Negamax에서는 이전 단계의 값에 음수를 취함으로써 어느 단계에서나 MinMax 함수를 적용할 수 있다. 따라서 Min, Max 함수를 번갈아 사용하지 않아도 되고 재귀함수를 이용하여 구현하는 것이 훨씬 적합하다.

2. 인공지능의 구현 방법

<인공지능실행방법>

프로그램을 실행하면 아래 그림과 같이 “Would you like to go first [y]?” 라고 뜨는데 y를 입력하면 사용자, 즉 Human 이 선공을 할 수 있고 n를 입력하면 Ai 가 선공을 한다.

AI의 돌은 0으로 표현되고 Human의 돌은 X로 표현된다.




```
do {
    print_board(width, height);
    if (player[turn] == HUMAN) {
        do {
            sprintf(buffer, "Drop in which column");
            move = get_num(buffer, 1, width, -1) - 1;
        } while (!c4_make_move(turn, move, NULL));
    }
    else {
        sprintf(buffer, "Heuristic(1)? Or Rule(2)");
        int mode = 0;
        //scanf("%d", &mode);
        while (mode != 1 && mode != 2) {
            printf("select 1 or 2\n");
            mode = get_num(buffer, 1, 2, 0);
            //                scanf("%d", &mode);
        }
        if (mode == 1) {
            printf("AI Thinking.");

            fflush(stdout);
            c4_auto_move(turn, level[turn], &move, NULL);

            printf("\n\nI dropped my piece into column %d.\n", move + 1);
        }
        else if (mode == 2) {
            printf("Rule Based\n");
            fflush(stdout);
            apply_rule(turn, &move, NULL);
            printf("\n\nI dropped my piece into column %d.\n", move + 1);

        }
```

-Human 차례일 때 : “Drop in which column” 라는 메세지가 뜨는데 1부터 7까지의 컬럼 번호를 입력하면 중력을 적용하여 보드에 돌을 둔다.

-AI 차례일 때: “Heuristic(1)? Or Rule(2)?” 라는 메세지가 뜨고 1을 입력하면 ai 가 negamax 를 적용한 휴리스틱 알고리즘으로, 2를 입력하면 우리가 구현한 rule을 바탕으로 돌을 둔 보드와 어떤 열에 돌을 두었다는 메시지가 콘솔창에 뜬다.

1)Heuristic

6*7 connect 4 보드에서 42개의 각 위치에 따라 가로, 세로, 대각선으로 승리하는 경우의 수는 본 코드 num_of_win_places(int x, int y, int n); 함수에서 결정한다.

```
static int
num_of_win_places(int x, int y, int n)
{
    if (x < n && y < n)
        return 0;
    else if (x < n)
        return x * ((y - n) + 1);
    else if (y < n)
        return y * ((x - n) + 1);
    else
        return 4 * x*y - 3 * x*n - 3 * y*n + 3 * x + 3 * y - 4 * n + 2 * n*n + 2;
}
```

x는 열, y는 행, n은 이기는 돌의 개수 조건이다.

connect4 의 경우에는 x = 7, y = 6, n = 4 으로 총 69가지이다.

임의의 보드 포지션에 돌을 놓았을 때, 알고리즘이 플레이어가 이길 수 있게 그 포지션에서 4개가 연결될 수 있는 가지 수를 받아 오는 것이다. 반대로 상대방한테는 더이상 거기에 가능성을 주면 안되는 장소이다. 따라서, 각 플레이어의 승리를 예측할 때, 69개의 승리방법을 더 많이 가지는 곳에 높은 점수를 부여한다. 그에 따른 AI의 점수와 상대방의 점수의 차이를 최종 점수로 반환한다.

본 코드에서는 최종 점수로 evaluate 함수 안에서 다음과 같이 반환한다. score는 drop_piece(돌을 선택하여 두는 것, 선택된 column에 얼마나 돌이 쌓아 있는 지 확인 후, 가장 위의 row 즉 놓을 수 있는 row를 반환하는 함수) 할 때마다 update_score 함수를 통해 map 배열에 저장된 각 위치마다의 4개를 만드는 가지 수를 받아온다. 밑의 map 배열의 함수선언과 배열의 값들을 살펴보면 가운데로 올 수록, 4를 만드는 가지수가 크다. 이를 통해, 점수가 update 되면서 가운데 컬럼에 가까운 순서대로 승리를 이끈다.

```
else if (level == depth)
    return goodness_of(player); //(current_state->score[player] - current_state->score[other(player)])
```

승부 결정 : win_coords 함수에서 보드의 북동쪽 절반, 보드의 남서쪽 절반에서 이기는 경우를 찾아서 4개가 연결된 좌표를 반환한다. 게임의 승부가 결정된 뒤 CONNECT4의 위치를 알려주는데에도 쓰인다. 4개가 되는 모든 경우를 반환하는 것이 아니라 각 구역에서 1개만 찾게되더라도 찾는 것을 멈추기 때문에 효율적이다.

```
void
c4_win_coords(int *x1, int *y1, int *x2, int *y2)
{
    register int i, j, k;
    int winner, win_pos = 0;
    bool found;

    assert(game_in_progress);

    winner = current_state->winner;
    assert(winner != C4_NONE);

    while (current_state->score_array[winner][win_pos] != magic_win_number)
        win_pos++;

    /* Find the lower-left piece of the winning connection. */

    found = false;
    for (j = 0; j<size_y && !found; j++)
        for (i = 0; i<size_x && !found; i++)
            for (k = 0; map[i][j][k] != -1; k++)
                if (map[i][j][k] == win_pos) {
                    *x1 = i;
                    *y1 = j;
                    found = true;
                    break;
                }

    /* Find the upper-right piece of the winning connection. */

    found = false;
    for (j = size_y - 1; j >= 0 && !found; j--)
        for (i = size_x - 1; i >= 0 && !found; i--)
            for (k = 0; map[i][j][k] != -1; k++)
                if (map[i][j][k] == win_pos) {
                    *x2 = i;
                    *y2 = j;
                    found = true;
                    break;
                }
}
```

	0	1	2	3	4	5	6
5	20,26,59	20,21,29,62	20,21,22,32,65	20,21,22,23,35,47,68	21,22,23,38,50	22,23,41,53	23,44,56
4	16,25,26,58	16,17,28,29,59,61	16,17,18,31,32,47,62,64	16,17,18,19,34,35,46,50,65,67	17,18,19,37,38,49,53,68	18,19,40,41,52,56	19,43,44,55
3	12,24,25,26,57	12,13,27,28,29,47,58,60	12,13,14,30,31,32,46,50,59,61,63	12,13,14,15,33,34,35,45,49,53,62,64,66	13,14,15,36,37,38,48,52,56,65,67	14,15,39,40,41,51,55,68	15,42,43,44,54
2	8,24,25,26,47	8,9,27,28,29,46,50,57	8,9,10,30,31,32,45,49,53,58,60	8,9,10,11,33,34,35,48,52,56,59,61,63	9,10,11,36,37,38,51,55,62,64,66	10,11,39,40,41,54,65,67	11,42,43,44,68
1	4,24,25,46	4,5,27,28,45,49	4,5,6,30,31,48,52,57	4,5,6,7,33,34,51,55,58,60	5,6,7,36,37,54,61,63	6,7,39,40,64,66	7,42,43,67
0	0,24,45	0,1,27,48	0,1,2,30,51	0,1,2,3,33,54,57	1,2,3,36,60	2,3,39,63	3,42,66

```
 0 - 23: horizontal wins
24 - 44: vertical wins
45 - 56: forward diagonal wins
57 - 68: backward diagonal wins
```

```
    map = (int ***)emalloc(size_x * sizeof(int **));
    for (i = 0; i<size_x; i++) {
        map[i] = (int **)emalloc(size_y * sizeof(int *));
        for (j = 0; j<size_y; j++) {
            map[i][j] = (int *)emalloc((num_to_connect * 4 + 1) * sizeof(int));
            map[i][j][0] = -1;
        }
    }

    win_index = 0;
```

map 배열에 저장되는 4개가 연결 될 수 있는 area에 label 을 부여한 것이다.

c4_new_game(int width, int height, int num) 함수를 통해 가로, 세로, 상승형 대각선, 하강형 대각선 모양의 CONNECT4의 개수가 각 column 마다 몇개가 있는지 계산하였고 이를 통해 가운데일 수록 높은 점수가 부여되었다. 따라서 가운데부터 점수를 계산하는 search algorithm을 사용하여 보다 더 알파베타 프루닝의 계산 속도를 빠르게 하였다.

```c
bool
c4_auto_move(int player, int level, int *column, int *row)
{
    int best_column = -1, goodness = 0, best_worst = -(INT_MAX);
    int num_of_equal = 0, real_player, current_column, result;

    assert(game_in_progress);
    assert(!move_in_progress);
    assert(level >= 1 && level <= C4_MAX_LEVEL);

    real_player = real_player(player);

    if (current_state->num_of_pieces < 1 &&
        size_x == 7 && size_y == 6 && num_to_connect == 4 &&
        (current_state->num_of_pieces == 0 ||
            current_state->board[3][0] != C4_NONE)) {
        //When you are the first player, you can't choose the middle of column
        if (column != NULL)
            *column = 2;
        if (row != NULL)
            *row = current_state->num_of_pieces;
        drop_piece(real_player, 2);
        return true;
    }

    if (current_state->num_of_pieces == 1) {
        //When you are the second player, you have to choose the middle
        if (column != NULL)
            *column = 2;
        if (row != NULL)
            *row = current_state->num_of_pieces;
        drop_piece(real_player, 3);
        return true;
    }
```

다음은 heuristic mode를 선택하면 돌을 두게 하는 c4_auto_move 함수이다. 위 코드에서 두 개의 큰 if 조건문은 과제의 기본 전제 조건을 나타낸다. 선공일 때는 가운데를 선점하지 못한다는 것을 반영한 것이다.

```c
    /* Otherwise, look ahead to see how good this move may turn out */
    /* to be (assuming the opponent makes the best moves possible). */
    else {
        next_poll = clock() + poll_interval;
        goodness = evaluate(real_player, level, -(INT_MAX), -best_worst);
    }

    /* If this move looks better than the ones previously considered, */
    /* remember it.                                                   */
    if (goodness > best_worst) {
        best_worst = goodness;
        best_column = current_column;
        num_of_equal = 1;
    }

    /* If two moves are equally as good, make a random decision. */
    else if (goodness == best_worst) {
        num_of_equal++;
        if ((rand() >> 4) % num_of_equal == 0)
            best_column = current_column;
    }

    pop_state();
}

move_in_progress = false;

/* Drop the piece in the column decided upon. */

if (best_column >= 0) {
    result = drop_piece(real_player, best_column);
    if (column != NULL)
        *column = best_column;
    if (row != NULL)
        *row = result;
    return true;
}
else
    return false;
}
```

```c
static int
evaluate(int player, int level, int alpha, int beta)
// c4_auto_make 함수안에서 goodness = evaluate(real_player, level, -(INT_MAX), -best_worst) 로 불리움;
{
    if (poll_function != NULL && next_poll <= clock()) {
        next_poll += poll_interval;
        (*poll_function)();
    }

    if (current_state->winner == player)
        return INT_MAX - depth;
    else if (current_state->winner == other(player))
        return -(INT_MAX - depth);
    else if (current_state->num_of_pieces == total_size)
        return 0; /* a tie */
    else if (level == depth)
        return goodness_of(player); //(current_state->score[player] - current_state->score[other(player)])
    else {
        /* Assume it is the other player's turn. */
        int best = -(INT_MAX);
        int maxab = alpha;
        for (int i = 0; i<size_x; i++) {
            if (current_state->board[drop_order[i]][size_y - 1] != C4_NONE)
                continue; /* The column is full. */
            push_state();
            drop_piece(other(player), drop_order[i]);
            int goodness = evaluate(other(player), level, -beta, -maxab); //recursive
            if (goodness > best) {
                best = goodness;
                if (best > maxab)
                    maxab = best;
            }
            pop_state();
            if (best > beta)
                break;
        }

        /* What's good for the other player is bad for this one. */
        return -best;
    }
```

c4_auto_move 에서 불러온 column의 점수를 판단하기 위해서 evaluate 함수를 불러온다. 이때, evaluate 함수에 넣어주는 인자 값은 다음과 같다.

```c
goodness = evaluate(real_player, level, -(INT_MAX), -best_worst);
```

이때 음수 값을 넣어주는 이유는 evaluate 함수가 negamax algorithm 방식으로 구현되어 max(a, b) == - min(-a, -b); 를 이용한 것이다. 그리고 함수 안에서 상대방의 점수를 계산하고 싶을 때, 자기 함수에 음수를 부여하여 재귀하는 방식으로 구현되었다.

```c
int goodness = evaluate(other(player), level, -beta, -maxab); //recursive
```

2)Rule Based

- apply_rule 함수에서 매 상황마다 7개의 컬럼에 두었을 때 각 컬럼에 대한 점수를 저장하는 변수(ruleflag[7])를 선언했다. 이 변수는 중간 컬럼으로 갈 수록 점수가 더 높게 초기화 되어 있다. 각 컬럼에 돌을 둬보고 점수가 가장 큰 컬럼에 돌을 둔다. (이 때 꽉 찬 컬럼일 경우 변수에 -300000이라는 작은 점수를 줘서 선택이 되지 못하게 한다.) ruleOfCol[][] 변수는 각 컬럼에서 적용된 룰을 저장하는 변수이다.

- eval_rule 함수에서는 apply_rule 에서 ruleOfCol[i] 를 받아온다. i 번째 컬럼에 적용된 룰을 저장하는 변수이다. rule[] 변수는 각 컬럼에 대한 점수를 저장한다. rule 변수의 값에 변화가 있으면 룰이 적용 된 것이므로 ruleOfCol[][]에 룰의 인덱스를 저장한다.

for문으로 칼럼에 돌을 둔 전체 보드의 상황에 대해 어떤 룰을 적용할 것인지 판단한다. for문을 돌면서 total 변수에 rule 변수 값을 계속 더하고 total을 반환한다.

```
int ruleflag[7];
//int rule_index;
int ruleOfCol[7][8] = { 0 };

void apply_rule(int player, int *column, int *row) {
    int max = -300001;
    int max_col = -100000;


    int real_player, result;

    assert(game_in_progress);
    assert(!move_in_progress);


    real_player = real_player(player);

    //중간칼럼 더 점수 높게
    for (int i = 0; i<4; i++) {
        ruleflag[i] = i + 5;
    }
    for (int i = 4; i<7; i++) {
        ruleflag[i] = ruleflag[6 - i];
    }
```

```
for (int i = 0; i<7; i++) {
    push_state();
    int numrow = drop_piece(real_player, i);
    if (numrow == -1) {
        ruleflag[i] = -300000;
        //      printf("ruleflag%d= %d₩n", i + 1, ruleflag[i]);
        pop_state();
        continue;
    }

    else {
        ruleflag[i] += eval_rule(ruleOfCol[i]);
        //      printf("ruleflag%d= %d₩n", i + 1, ruleflag[i]);
        pop_state();

    }
}

for (int i = 0; i<7; i++) {
    if (max<ruleflag[i]) {
        max = ruleflag[i];
        max_col = i;
    }
}

result = drop_piece(real_player, max_col);
```

```
int eval_rule(int nthCol[]) {
    int score = 0;
    int total = 0;
    int rule[8];
    for (int k = 0; k < 8; k++) {
        rule[k] = 0;
    }
    for (int i = 0; i <= 6; i++) {//행
        for (int j = 0; j <= 5; j++) {//렬
                                /*rule 1*/  //AI 가 4줄이 되면 바로 놓는다.
            if (i < 4 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1) {
                /*   if (i > 0 && current_state->board[i][j-1] != C4_NONE && current_state->board[i + 1][j - 1] != C4_NONE && current_state->board[i + 2][j - 1] != C4_NONE && current_state->board[i + 3][j - 1] !=
                //가운*/
                score = 5000000;
                rule[0] += score;
                //   printf("1-1₩n");
                //
```

```
    for (int l = 0; l < 8; l++) {
        total += rule[l];
        nthCol[l] = 0;
        if (rule[l] != 0) {
            nthCol[l] = l + 1;
            }
        }


    return total;
}
```

<룰 설명>

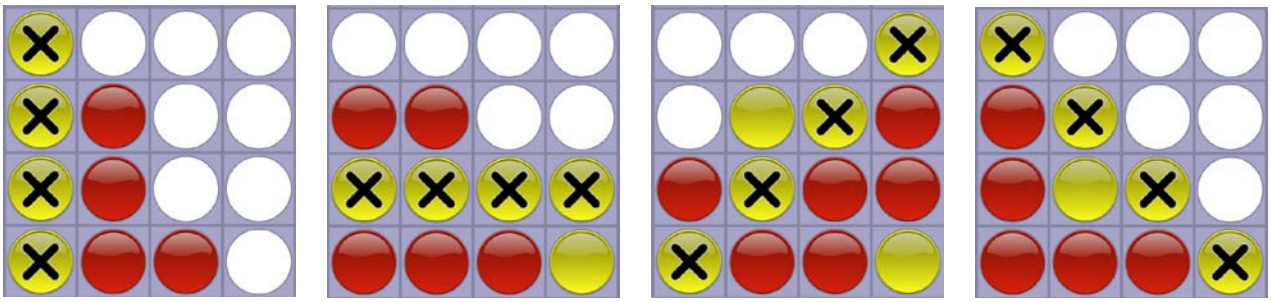
(참고사항 : 상황 설명을 위한 첨부되는 그림에서 노랑색 : AI / 빨간색 : 상대방)

rule 0. 선공에는 가운데를 선점하지 못한다. / 후공일 때 2회까지 무조건 가운데를 선점한다.

```
479     if (current_state->num_of_pieces < 1) {
480         if (column != NULL)
481             *column = 2;
482         if (row != NULL)
483             *row = current_state->num_of_pieces;
484         drop_piece(real_player, 2);
485         printf("I chosed the rule 0₩n");
486         return;
487     }
```

```
489     else if (current_state->num_of_pieces < 4) {
490         if (column != NULL)
491             *column = 3;
492         if (row != NULL)
493             *row = current_state->num_of_pieces;
494         drop_piece(real_player, 3);
495         printf("I chosed the rule 0₩n");
496         return;
497     }
```

rule 1. AI 가 connect4 를 만드는 자리에 둔다.

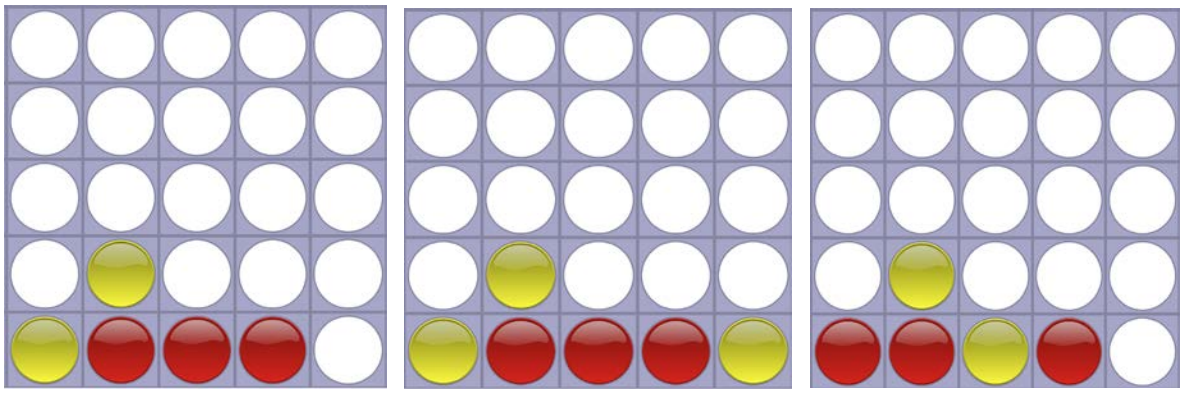


.

위의 그림과 같이 가로, 세로, 올라가는 대각선, 내려가는 대각선 모두에 대해 AI의 돌이 4개가 될 가능성이 있으면 룰 중 가장 큰 점수인 5백만을 준다.

```
score = 5000000;
rule[0] += score;
```

rule 2. 상대방이 connect4를 만드는 자리 막기



#2-1 상대방의 돌이 연속으로 3개가 있을 경우)

1) 상대방의 연속된 돌의 양 옆 중 하나만 막았을 경우에는 250000점을 주었다.
2) 상대방의 연속된 돌의 양 옆 모두 막았을 경우에는 300000 + 2500000, 총 550000점을 주었다.

2)의 경우에 점수를 더 부여한 이유는 전단계에서 1)의 경우가 있었을 때, AI가 이것으로 만족하고 반대쪽을 막지 않았기 때문이다.

```
if (i < 4 && current_state->board[i][j] == 1 && current_state->board[i+1][j] == 0 && current_state->board[i+2][j] == 0 &&
    current_state->board[i+3][j] == 0) {
    if (i < 3 && current_state->board[i+4][j] == 1) {
        score = 300000;
        rule[1] += score;
    }
    score = 250000;
    rule[1] += score;
    //horizontal OXXX_(_!=X)
}
```

#2-2 상대방의 돌이 XXOX 혹은 XOXX 모양일 경우)

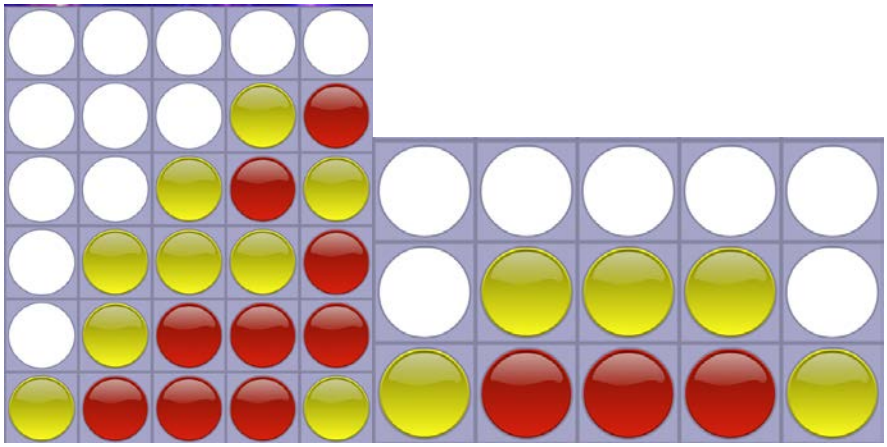
```
if (i < 4 && current_state->board[i][j] == 0 && current_state->board[i+1][j] == 0 && current_state->board[i+2][j] == 1 &&
    current_state->board[i+3][j] == 0) {
    //  if (current_state->board[i +2][j-1] != C4_NONE) {
    score = 250000;
    rule[1] += score;
    //  printf("2-5\n");
    //   }
    //horizontal XXOX
}
```

점수를 원래는 백단위로 해주었으나 모든 룰에서 2번룰이 바로 지는 것을 막아주는 룰이기 때문에 다른 룰들의 누적 점수가 아무리 커도 2번 룰의 점수보다는 작게 하기 위해 10만 단위로 점수를 부여하였다. 또한 가로 뿐만 아니라 세로, 상승형 대각선, 하강형 대각선 경우를 모두 추가하였다.

rule 3. AI가 connect3를 만드는 자리에 두기 - 가로, 세로, 올라가는 대각선, 내려가는 대각선

#3-1 양쪽이 비어 있고 빈 칸의 받침이 있는 connect3를 만들 경우)

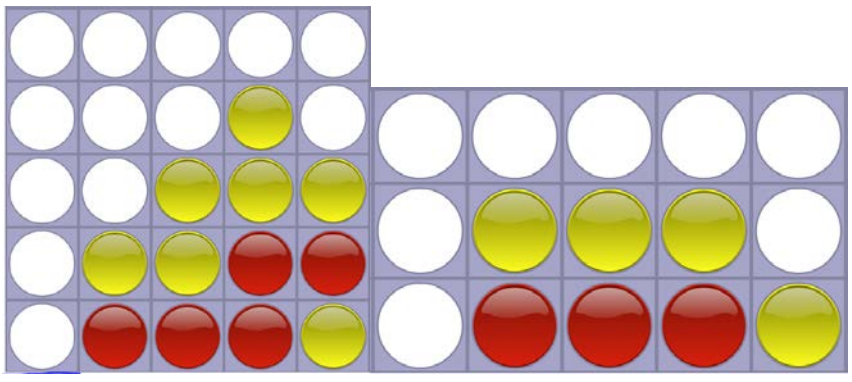
```
        //_OOO_ (both full) horizontal
if (i < 4 && i > 0 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i - 1][j] == C4_NONE && current_state->board[i + 3][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 3][j - 1] != C4_NONE) {
        score = 10000;
        rule[2] += score;
        //      printf("3-1\n");
    }
    else if (j == 0) {
        score = 10000;
        rule[2] += score;
    }
}
```



위 그림과 같이 양쪽이 비어있고 양 빈칸의 받침이 있을 경우 connect3을 만들면, 다음 판에 바로 이길 수 있으므로 큰 점수 10000을 주었다. 가로, 세로 두 대각선 모두에 대해 정의 하였고 첫줄일 경우 빈칸의 받침이 다 있는 경우이므로 y==0이라는 조건만 추가하여 마찬가지로 높은 점수를 주었다.

#3-2 양쪽이 비어 있고 빈 칸의 받침이 한 쪽만 있는 connect3를 만들 경우)

```
//_OOO_ (XOR full) horizontal
if (j > 0 && i < 3 && current_state->board[i][j] == C4_NONE && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1 && current_state->board[i + 4][j] == C4_NONE) {
    if (current_state->board[i][j - 1] == C4_NONE && current_state->board[i + 4][j - 1] != C4_NONE) {
        score = 1000;
        rule[2] += score;
        //      printf("3-3\n");
    }
    else if (current_state->board[i][j - 1] != C4_NONE && current_state->board[i + 4][j] == C4_NONE) {
        score = 1000;
        rule[2] += score;
        //      printf("3-3\n");
    }
}
```



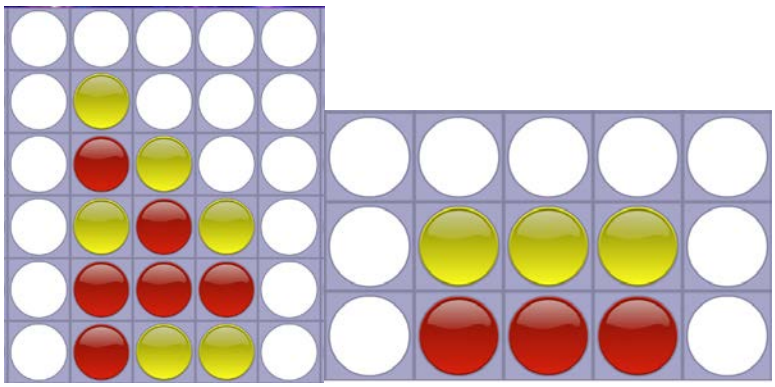
왼쪽의 그림은 왼쪽 빈칸의 받침이 있는 경우이고 오른쪽의 그림은 오른쪽 빈칸의 받침이 있는 경우이다.

이 경우에는 한쪽 빈칸의 받침이 없어 받침이 있는 쪽에 상대방이 돌을 놓아버리면 바로 connect4를 만들지 못하기 때문의 앞의 경우보다는 점수가 작은 1000을 주었다. 가로의 경우에는 첫 줄을 생각할 필요가 없지만 대각선의 경우, 첫 줄에 한쪽 빈칸이 있으면 당연히 빈칸의 한쪽 받침이 있는 경우이므로 따로 생각해주었다.

```
//_000_ (both empty) horizontal
if (j > 0 && i < 3 && current_state->board[i][j] == C4_NONE && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1 && current_state->board[i + 4][j] == C4_NONE) {
    if (current_state->board[i][j - 1] == C4_NONE && current_state->board[i + 4][j - 1] == C4_NONE) {
        score = 2000;
        rule[2] += score;
        //        printf("3-4₩n");

    }
}
```

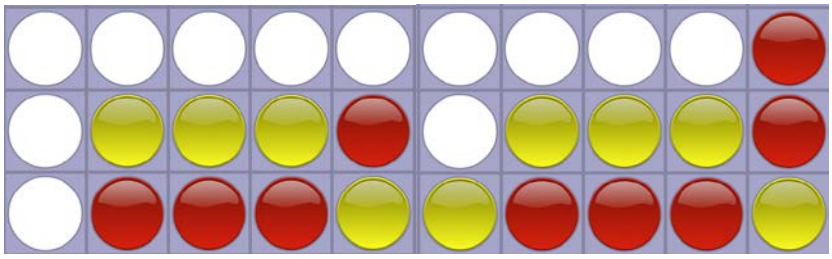
#3-3 양쪽이 비어 있고 빈칸의 받침이 모두 없는 경우)

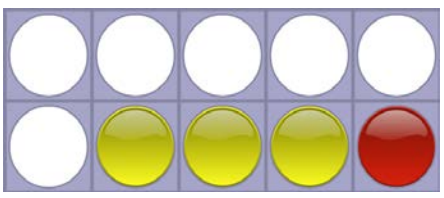


양쪽의 빈칸의 받침이 모두 없는 경우, 상대방이 빈칸의 열에 두면 connect4를 만들 수 있는 가능성이 높아지므로 앞의 경우보다 높은 2000점을 부여하였다.
이 경우는 첫 줄일 경우가 없으므로 구현하지 않았다.

#3-4 한쪽이 상대방 돌 또는 벽으로 막혀 있고 다른 한쪽은 비어 있는 경우)

```
//_000X (_below empty) horizontal
if (i < 3 && current_state->board[i][j] == C4_NONE && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1 && current_state->board[i + 4][j] == 0) {
    if (j > 0 && current_state->board[i][j - 1] == C4_NONE) {
        score = 1000;
        rule[2] += score;
        //        printf("3-5₩n");
    }
    else if (j > 0 && current_state->board[i][j - 1] != C4_NONE) {
        score = 300;
        rule[2] += score;
    }
    else if (j == 0) { // firstline
        score = 300;
        rule[2] += score;
    }
}
```



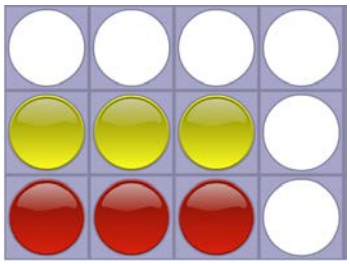


1) 빈칸의 받침이 없는 경우 : 상대방이 빈칸이 있는 열에 둘 경우 AI가 connect4를 만들 가능성이 높아지므로 1000점을 부여한다.
2) 빈칸의 받침이 있는 경우 (=첫 줄) : 이 경우 상대가 빈칸에 둬버리면 바로 막히기 때문에 이전 경우보다 작은 점수 300점을 부여한다.

아래의 코드와 그림은 벽으로 막혀 있는 경우이다.

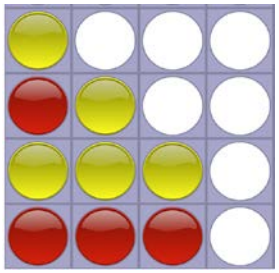
```
//_OOOwall (_below empty) horizontal
if (i == 3 && current_state->board[i][j] == C4_NONE && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 ,
    if (j > 0 && current_state->board[i][j - 1] == C4_NONE) {
        score = 1000;
        rule[2] += score;
        //      printf("3-6₩n");
    }
    else if (j > 0 && current_state->board[i][j - 1] != C4_NONE) {
        score = 300;
        rule[2] += score;
    }
    else if (j == 0) { // firstline
        score = 300;
        rule[2] += score;
    }

}
```



<= 빈칸의 받침이 없는경우

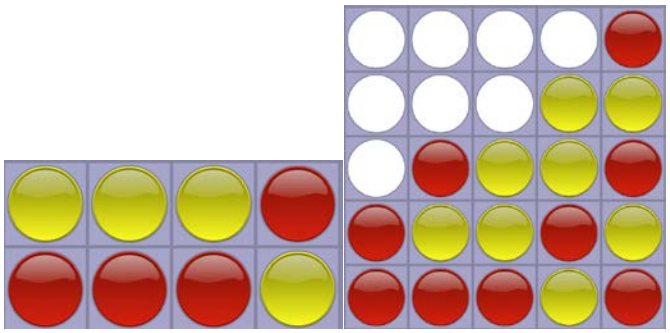
빈칸의 받침이 있는 경우=>



코드를 짤 때 빈칸의 받침이 있는 경우와 없는 경우를 따로 생각해서 구현했는데 후에 합칠 수 있음을 깨닫고 합쳤다.

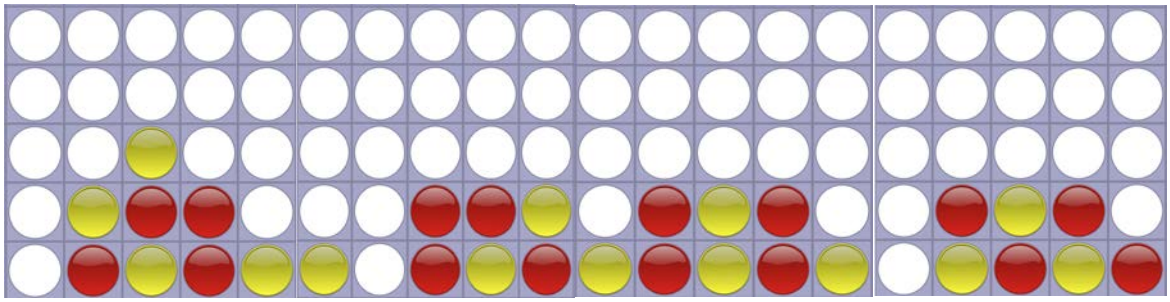
#3-5 양쪽 다 막혀 있는 connect4를 만들 경우)

```
,
// XOOOX horizontal
if (i < 3 && current_state->board[i][j] == 0 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1 && current_state->board[i + 4][j] == 0) {
    score = -200;
    rule[2] += score;
}
//wallOOOX
if (i ==0 &&current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 0) {
    score = -200;
    rule[2] += score;
}
//XOOOwall
if (i ==3 && current_state->board[i][j] == 0 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i + 3][j] == 1) {
    score = -200;
    rule[2] += score;
}
```



AI가 connect3을 만들었을 때 양쪽이 벽이나 상대방 돌로 막혀 있다면 connect4를 만들지 못하므로 두는 중요성이 떨어진다. 따라서 점수를 -200을 부여한다.

rule 4. 상대방이 connect3를 만드는 자리 막기



rule 4의 경우 상대방이 연속된 3개의 돌을 만들 수 있을 때 그 양 옆이 빈칸임을 전제하고 만든 rule이다. 만약 한 쪽만 빈칸일 경우 (예를 들면 OXX_) 막지 않더라도 큰 위협이 되지 않기 때문이다. 하지만 양 옆이 모두 빈칸일 경우 지금 막지 않으면 다음 턴에 상대방이 양 옆이 빈칸인 연속된 세개의 돌을 만들 수 있기 때문에 다음 턴에 우리가 반드시 지게되는 결과를 초래할 수 있다. 또한 가로 뿐만 아니라 세로, 상승형 대각선, 하강형 대각선 경우를 모두 추가하였다.

#4-1 상대방의 돌이 2개 연속으로 있을 때)

1) 상대방 돌의 양 옆 빈칸의 받침이 모두 있을 경우
2) 상대방 돌의 양 옆 빈칸의 받침이 한쪽에만 있을 경우

1)의 경우 이것을 막지 않을 경우 어느 쪽으로든 상대방이 바로 3개를 만들 수 있기 때문에 다음 턴에 반드시 지게 되므로 5000점을 부여하였다. 하지만 2)의 경우 내가 막게 되면 상대방이 다음번에 바로 공격은 할 수 없기 때문에 그보다 훨씬 적은 150점만 부여하였다. 아래 j == 0은 첫번째 줄, 즉 양 옆 빈칸의 받침이 모두 있을 경우와 같기 때문에 5000점을 부여하였다.

```
if (i > 0 && i<4 && current_state->board[i][j] == 0 && current_state->board[i + 1][j] == 0 && current_state->board[i + 2][j]
    == 1 && current_state->board[i - 1][j] == C4_NONE && current_state->board[i + 3][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 3][j - 1] != C4_NONE) {
        score = 5000;
        rule[3] += score;
    }
    else if (j > 0 && current_state->board[i - 1][j - 1] == C4_NONE && current_state->board[i + 3][j - 1] != C4_NONE) {
        score = 150;
        rule[3] += score;
    }
    else if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 3][j - 1] == C4_NONE) {
        score = 150;
        rule[3] += score;
    }

    else if (j == 0) {
        score = 5000;
        rule[3] += score;
    }
    //    printf("4-1\n");
}
```

#4-2 상대방의 돌이 빈칸 하나를 두고 양옆에 있을 때) (XOX)

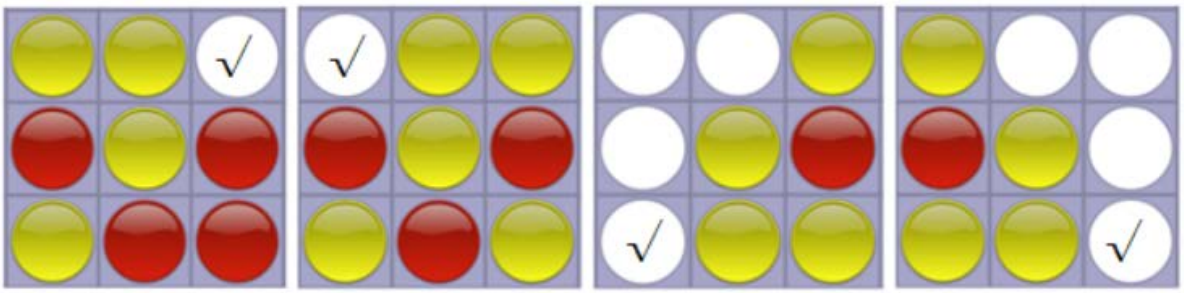
1) 상대방의 돌의 양 옆 빈칸의 받침이 모두 있을 경우
2) 상대방의 돌의 양 옆 빈칸의 받침이 한쪽에만 있을 경우

#4-1와 동일하다.

```
if (i > 0 && i<4 && current_state->board[i][j] == 0 && current_state->board[i + 1][j] == 1 && current_state->board[i + 2][j]
    == 0 && current_state->board[i - 1][j] == C4_NONE && current_state->board[i + 3][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 3][j - 1] != C4_NONE) {
        score = 5000;
        rule[3] += score;
    }
    else if (j > 0 && (current_state->board[i - 1][j - 1] != C4_NONE || current_state->board[i + 3][j - 1] != C4_NONE)) {
        score = 150;
        rule[3] += score;
    }
    //      printf("4-5\n");
    else if (j == 0) {
        score = 5000;
        rule[3] += score;
    }
}
```

rule 5. 7자 모양

7자는 한 번의 돌을 두는 실행을 통해 33 모양을 만들 수 있는 매우 중요한 형태이다. 가로(horizontal)과 대각선(diagonal)의 connect3을 만들기 때문에, connect4에 쉽게 도달하기 좋은 모양이다. 그래서 AI가 7자를 만들 수 있는 경우에는, 높은 점수를 주어 그 곳에 착수하도록 하였고, 상대방이 7자를 만드는 경우는 질 수 있는 크리티컬한 상황이기에, 7자 형성을 방해하도록 그 곳에 착수하도록 점수를 부여하였다. 7자의 모양이 생기는 경우는 크게 네 가지로, 기본 7자 모양 / 좌우 반전 7자 / 상하 좌우 반전 7자 / 상하 반전 7자가 있다.

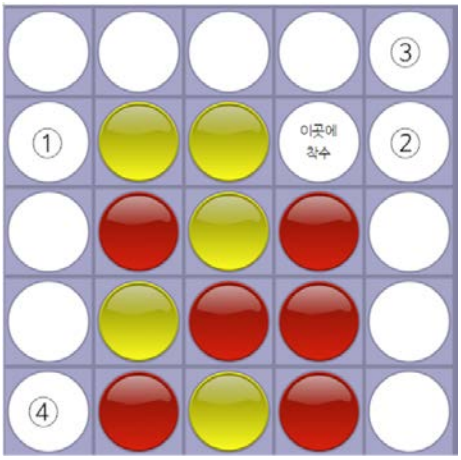


다음 4개의 그림이 4가지 경우를 표현한 그림이다. 7자의 꼭짓점을 둠으로써, 3개가 동시에 두개 완성되는 것이기에, 꼭짓점을 착수하는 경우만 생각하였다. 7자의 꼭짓점은 그림에서 체크된 부분이다.

```
/*rule 5*/ //7자 모양
if (i>1 && j>1 && current_state->board[i][j] == 1 && current_state->board[i - 1][j] == 1&& current_state->board[i - 2][j] == 1
    && current_state->board[i - 1][j - 1] == 1 && current_state->board[i - 2][j - 2] == 1) {

    if (i>2 && i < 6 && j < 5 && current_state->board[i + 1][j + 1] == 0 && current_state->board[i - 3][j] == 0) {
        //4개 만들 가능성이 막혀 있으면 점수 주지 마라!
        score = -100;
        rule[4] += score;
        //    printf("5-1\n");
    }
    else {
        score = 2000;
        rule[4] += score;
        //  printf("5-1\n");
    }
    //내가 7만들수 있는거 // 7
}
```

다음은 7자 모양일 때, 착수하도록 지시하는 rule 5 의 일부분이다. 전체 if 문의 괄호 안에는 꼭지점(board[i][j])을 기준으로 옆, 대각선의 값을 읽어서 그 때 모양이 7을 만족하면 점수를 받는 방식이다. 전체 if 안에서 다시 if / else 문으로 나뉘는데 이 조건은 7자를 만들어도 4를 만들 수 있는 옆의 위치가 막혀 있어서 소용이 없는 경우에는 음수를 주어 착수하지 못하도록 만드는 것이다.

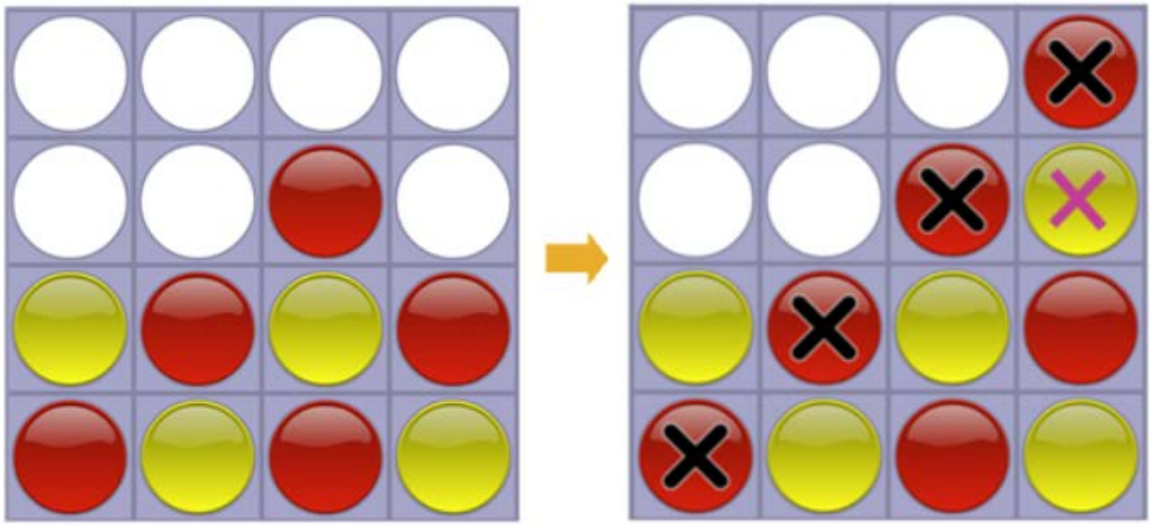


다음 그림을 살펴보면, (이곳에 착수) 위치에 착수를 하면 ①②③④ 부분이 connect 4를 만들 수 있는 가능성을 지닌 곳이다. ①②③④의 부분이 상대방의 돌이라면, 7자를 만들어도 크리티컬 하지 않기 때문에 이런 경우를 제외하도록 수정하였다. 처음에 4를 만드는 가능성의 막힘 여부를 고려하지 않고 코드를 작성하였다가, 여러 차례 실행을 하는 과정에서, 필요 없는 7자에 착수하는 경우가 발생하여 시행착오를 겪은 후 올바르게 작동하도록 고칠 수 있었다. ①②③④의 경우, ①③ 이 ②④의 위에 있고 CONNECT4 게임의 특성상, 밑의 row가 차 있어야, 위에 착수할 수 있기에 막혀 있는 경우는 위에 놓여진 ①③ 부분만 확인하였다. 기본 7자 뿐 아니라 나머지 4가지 경우에 대해서도 똑같이 적용하여 점수를 배분하였다.

상대방의 돌이 7자를 형성할 때에도, 그 돌의 옆이 막혀 있으면 4개를 만들 수 있는 가능성이 적기에 착수하지 않아도 되어 같은 방식으로 코드를 작성 후 점수를 올바르게 배분하였다.

rule 6. 두면 다음에 진다

다음 룰은 이번 턴에 AI가 착수한 돌로 인해, 상대방이 4개를 연결할 수 있는 가능성을 가져서 다음턴에 바로 지게 되는 상황을 막기 위한 룰이다. 4개가 만들어지기 전 상황은 xxx_ , xx_x , x_xx , _xxx 로 4가지 경우가 있다. (x는 상대방의 돌이고, _ 는 빈칸을 의미한다.) 또한, 이러한 4가지 경우에 대해 horizontal / diagonal ascending / diagonal descending 총 3가지 경우가 있다. 그래서 총 12가지 경우에 대해 코드로 구현하였다.



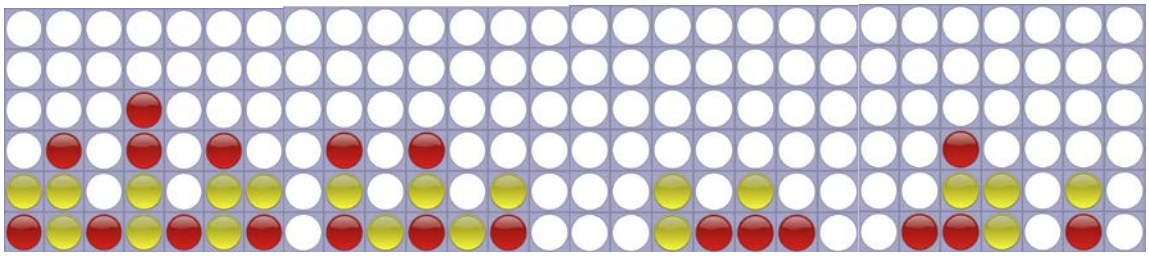
다음 그림은 diagonal ascending 중에서 xxx_ 경우의 그림이다. 상대방이 연속된 3개를 착수한 상황에서 AI가 분홍색 X 표시 위치에 착수하게 되면, 상대방의 받침이 되어 4개를 만들 수 있게 도와준 꼴이 되어버린다. 게임 특성상 원하는 위치에 바로 둘 수 없고, Column에서 row가 그 밑 까지 채워져야 둘 수 있기에, 우리가 먼저 두지 않도록 하는 것이다. 이 룰은 막지 않으면 다음 턴에 바로 지는 굉장히 중요한 상황이기에 매우 큰 절대값을 가진 음수의 점수를 부여하여 절대 이 위치에 AI가 착수하지 못하도록 코드를 작성하였다.

```
if (j < 5 && j>1 && i>2 && current_state->board[i - 3][j - 2] == 0 && current_state->board[i - 2][j - 1] == 0
    && current_state->board[i - 1][j] == 0 && current_state->board[i][j] == 1 &&
    current_state->board[i][j + 1] == C4_NONE) {//xxx_ diagonal ascending
    score = -10000;
    rule[5] += score;
    //      printf("6-10\n");
}
```

다음 코드는, 위의 상황인 diagonal ascending을 표현한 조건이다. board[i][j]가 AI가 두는 돌 위치이고 올라가는 대각선 위치에 상대방 돌인 0이 있는지 확인하는 조건문이다.

board[i][j+1] == C4_NONE 이라는 조건은 착수하는 지점의 위의 곳이 막혀 있으면 착수가 불가능 하기에 추가한 조건이다. 전체 코드 구현 방식이 돌을 둔 보드를 고정하고 전체 42개 칸을 읽는 것이기 때문에, 오류를 방지하기 위하여 추가하였다.

rule7. 돌 1개를 두는 것



#7-1 AI 돌 하나를 기준으로 양 쪽으로 모두 빈칸이고, 한 칸 띄우고 AI 돌 2개가 있을 때)

1) 2개의 빈칸 모두 받침이 있을 때
2) 2개의 빈칸 중 하나만 받침이 있을 때
3) 2개의 빈칸 모두 받침이 없을 때

1)의 경우 상대방이 하나의 빈칸에 돌을 두더라도 바로 다음 턴에 AI가 다른 하나의 빈칸에 돌을 두어 4개를 만들 수 있기 때문에 10000점을 주었다. 첫번째 줄의 경우도 이에 해당하므로 10000점을 주었다. 하지만 2)의 경우 상대방이 빈칸에 돌을 두면 바로 다음 턴에 4개를 만들 수 없으므로 1000점을 주었다. 3)의 경우 다음 턴에 빈칸 위에 돌을 둘 수 있기 때문에 막히게 되므로 500점을 주었다..

```
if (i < 4 && i > 2 && current_state->board[i][j] == 1 && current_state->board[i - 2][j] == 1 && current_state->board[i + 2]
    [j] == 1 && current_state->board[i - 3][j] == 1 && current_state->board[i + 3][j] == 1 && current_state->board[i - 1][j]
    == C4_NONE && current_state->board[i + 1][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 1][j - 1] != C4_NONE) {
        score = 10000;
        rule[6] += score;
    } //양쪽 받침 있다.

    else if (j == 0) { // first line
        score = 10000;
        rule[6] += score;
    }

    else if (j > 0 && (current_state->board[i - 1][j - 1] == C4_NONE && current_state->board[i + 1][j - 1] == C4_NONE)) {
        score = 1000;
        rule[6] += score;
    } //양쪽받침없다
    else if (j > 0 && (current_state->board[i - 1][j - 1] != C4_NONE || current_state->board[i + 1][j - 1] != C4_NONE)) {
        score = 500;
        rule[6] += score;
    } //한쪽받침있다

}
```

#7-2 AI 돌 하나를 기준으로 양 쪽으로 모두 빈칸이고, 한 칸 띄우고 AI 돌 1개가 있을 때)

1) 2개의 빈칸 모두 받침이 있을 때
2) 2개의 빈칸 중 하나만 받침이 있을 때
3) 2개의 빈칸 모두 받침이 없을 때

1)의 경우 상대방이 하나의 빈칸에 돌을 두더라도 바로 다음 턴에 AI가 다른 빈칸에 돌을 두어 한쪽이 막힌 3개를 만들 수 있기 때문에 200점을 주었다. 첫번째 줄도 이에 해당하므로 200점을 주었다. 하지만 2)의 경우 상대방이 빈칸에 돌을 두면 바로 다음 턴에 3개를 만들 수 없으므로 150점을 주었다. 3)의 경우 다음 턴에 상대방이 방어를 할 수 없지만, 나 역시 3개를 만들 수 없으므로 150점을 주었다.

```
if (i < 5 && i>1 && current_state->board[i][j] == 1 && current_state->board[i - 2][j] == 1 && current_state->board[i + 2][j]
    == 1 && current_state->board[i - 1][j] == C4_NONE && current_state->board[i + 1][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 1][j - 1] != C4_NONE) {
        score = 200;
        rule[6] += score;
    } //양쪽 받침 있다.
    if (j == 0) {
        score = 200;
        rule[6] += score;
    }
    else if (j > 0 && (current_state->board[i - 1][j - 1] == C4_NONE && current_state->board[i + 1][j - 1] == C4_NONE)) {
        score = 150;
        rule[6] += score;
    } //양쪽받침없다
    else if (j > 0 && (current_state->board[i - 1][j - 1] != C4_NONE || current_state->board[i + 1][j - 1] != C4_NONE)) {
        score = 100;
        rule[6] += score;
    } //한쪽받침있다

}/*
```

#7-3 빈칸 하나를 기준으로 양 쪽으로 모두 AI의 돌이고, AI 돌 좌우가 모두 빈칸일 때)

1) 가운데 빈칸의 받침이 있을 때
2) 가운데 빈칸의 받침이 없을 때

1) 의 경우 상대방이 바로 다음 턴에 가운데 빈칸에 돌을 두어 연속된 3개를 만들 수 없을 수 있기 때문에 150점을 부여하였다. 첫번째 줄도 1)에 포함되므로 같은 점수를 주었다. 2)의 경우 상대방이 바로 다음 턴에 가운데 빈칸에 돌을 둘 수 없지만 AI도 바로 다음 턴에 연속된 3개를 만들 수는 없기 때문에 1)보다 조금 높은 200점을 주었다.

```
if (i > 0 && i < 4 && current_state->board[i][j] == 1 && current_state->board[i + 2][j] == 1 && current_state->board[i - 1]
    [j] == C4_NONE && current_state->board[i + 1][j] == C4_NONE && current_state->board[i + 3][j] == C4_NONE) {
    if (j > 0 && current_state->board[i + 1][j - 1] != C4_NONE) {
        score = 150;
        rule[6] += score;
    } //가운데 받침 0
    else if (j == 0) {
        score = 150;
        rule[6] += score;
    }
    else {
        score = 200;
        rule[6] += score;
    }
}
```

#7-4 빈칸 하나를 기준으로 한 쪽으로 2개의 AI돌이 있고, 다른 한 쪽으로는 1개의 AI 돌이 있을 때)

1) 빈칸의 받침이 있을 때

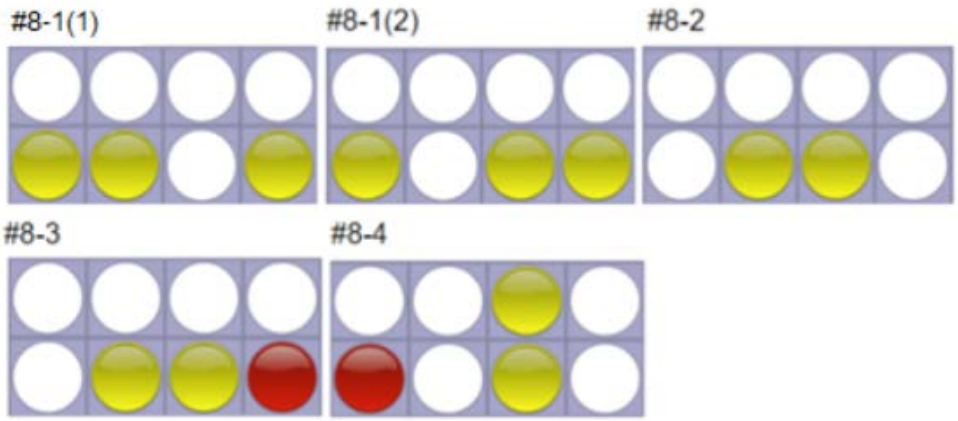
2) 빈칸의 받침이 없을 때

1) 의 경우 상대방이 바로 다음 턴에 가운데 빈칸에 돌을 두어 연속된 4개를 만들 수 없을 수 있기 때문에 300점을 부여하였다. 첫번째 줄도 1)에 포함되므로 같은 점수를 주었다. 2)의 경우 상대방이 바로 다음 턴에 가운데 빈칸에 돌을 둘 수 없지만 AI도 바로 다음 턴에 연속된 4개를 만들 수는 없기 때문에 1000점을 주었다. #7-3과 비교하였을 때 훨씬 높은 점수를 준 이유는 승리 결과에 더 가깝기 때문이다.

```
if (i>2 && current_state->board[i][j] == 1 && current_state->board[i - 2][j] == 1 && current_state->board[i - 3][j] == 1 &&
    current_state->board[i - 1][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE) {
        score = 300;
        rule[6] += score;
    }//받침있을 때
    else if (j == 0) {
        score = 300;
        rule[6] += score;
    }
    else {
        score = 1000;
        rule[6] += score;
    }
}
```

또한 #7-1 ~ #7-4의 모든 경우에 가로 뿐만 아니라 세로, 상승형 대각선, 하강형 대각선 경우를 모두 추가하였다.

rule 8. connect2를 만드는 것) (여기 두면 AI의 돌이 2개)



돌을 착수하여서 연속된 2개를 만드는 상황에 관한 룰이다. 이런 경우에는 크게 네 가지 경우가 있다. 첫번째 경우는 oo_o , o_oo 와 같이 한칸 띄어서 나의 돌이 있을 경우이다. 두번째 경우는 _oo_ 같이 양쪽이 비어있어서 4개를 만들 가능성이 있는 경우이다. 세번째 경우는 _oo? (? : blocked) 와 같이 한쪽 벽이 막혀 있어서 두번 째 경우보다는 4개를 만들 가능성이 적은 경우이다. 세번째 경우에서 (왼쪽이 막힌 경우 || 오른쪽이 막힌 경우) 로 생각하여, 양쪽이 다 막힌 경우까지 검사하도록 구현하였다. 마지막 경우는 단순하게 oo 같이 연속된 2개의 돌을 만드는 경우이다.

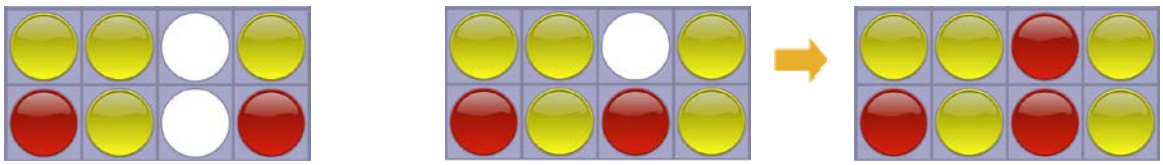
#8-1 한칸 띄어서 AI 돌이 있는 경우)

rule 8-1의 경우에 해당하는 모양은 #8-1(1)과 #8-1(2) 두가지 경우이다. 각 경우에 따라, horizontal / diagonal ascending / diagonal descending 세 가지 경우가 생겨서 크게 보아 총 6가지 경우에 대하여 룰을 작성하였다.

```
//00_0 이렇게 한칸 띄어서 본인돌이 있을 때
//horizontal
if (i < 4 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 &&
    current_state->board[i + 3][j] == 1 && current_state->board[i + 2][j] == C4_NONE) {
    //빈칸 밑에 받침이 없다.
    if (j>0 && current_state->board[i + 2][j - 1] == C4_NONE) {
        score = 1000;
        rule[7] += score;
        //printf("20_1\n");
    }
    //빈칸 밑에 받침이 있다. -> 바로 막혀서 점수 작게 줌.
    else if (j>0 && current_state->board[i + 2][j - 1] != C4_NONE) {
        score = 300;
        rule[7] += score;
    }
    else if (j == 0) {
        score = 300;
        rule[7] += score;
    }
}
```

다음 코드는 #8-1(1) oo_o 의 horizontal 경우이다. _ 빈칸 밑에 받침 여부에 따라 oo_o 모양의 중요도가 달라진다. 받침 여부를 전체 if 코드 안에서 if / else if / else if 구문으로 나누어 표현하였다. _ 빈칸 밑에 받침이 있다면, 다음 턴에 상대방의 우리의 4개 만드는 것을 방해하기 위하여 바로 막을 수 있기 때문에 점수를 작게 주었다. 그러나, _ 빈칸 밑에 받침이 없다면, 미래에 4개를 만들 수 있는 가능성이 있고, 상대방이 다음 턴에 바로 막지 못하기 때문에 상대적으로 큰 점수를 부여하였다. 마지막 j==0 인 경우는 첫 줄인 경우라 받침에 관한 조건이 필요 없어 따로 작성하였다. 첫 줄인 경우에도 바로 상대방에게 막힐 가능성이 있기에 낮은 점수를 부여하였다.

빈칸 밑에 받침이 없는 경우)　　빈칸 밑에 받침이 있는 경우)

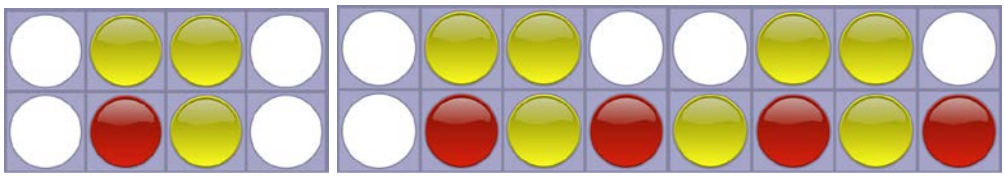


바로 막히지 않아 점수 1000점 부여　　/　　바로 막힐 수 있어서 낮은 점수 300점 부여

#8-2 양쪽이 비어 있어 4개를 만들 수 있는 경우)

양쪽이 비어 있어 돌 2개를 만들면 미래에 4개를 만들 수 있는 가능성이 있는 경우에 점수를 주는 것이다. 그러나 돌 2개를 만드는 것이기 때문에, 돌 3개를 만드는 룰보다는 작게 부여하고 여러번 시행착오 끝에 적절한 값의 크기를 부여하였다.

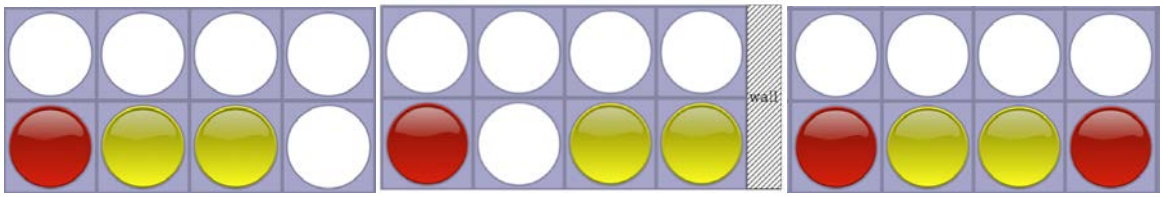
#8-2(1)　　#8-2(2)　　#8-2(3)



룰 8-2 의 경우 기본 전제가 양쪽이 비어 있는 것이고, 양쪽 빈칸의 밑이 empty 이냐 아니냐에 따라 점수가 조금씩 달라진다. 위의 경우같이 세가지 경우가 존재한다. 둘다 not empty 인 경우는 2수 안에 막힐 가능성이 있기에 점수를 가장 적게 주었다. 양쪽이 empty 인 경우가 막힐 가능성이 가장 적어 점수를 크게 부여하였다.

```
//양쪽 비어 있어 4개 만들 가능성 있다. _oo_
//horizontal
if (i < 5 && i>0 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 && current_state->board[i - 1][j] == C4_NONE
    && current_state->board[i + 2][j] == C4_NONE) {
    if (j > 0 && current_state->board[i - 1][j - 1] == C4_NONE && current_state->board[i + 2][j - 1] == C4_NONE) {
        // 빈칸의 빈칸 두개 모두 empty
        score = 150;
        rule[7] += score;
    }
    else if (j > 0 && ((current_state->board[i - 1][j - 1] == C4_NONE && current_state->board[i + 2][j - 1] != C4_NONE) ||
        (current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 2][j] == C4_NONE))) {
        // 빈칸의 빈칸 두개 중 하나가 empty
        score = 100;
        rule[7] += score;
    }
    else if (j > 0 && current_state->board[i - 1][j - 1] != C4_NONE && current_state->board[i + 2][j - 1] != C4_NONE) {
        // 빈칸의 빈칸 두개 모두 not empty
        score = 50;
        rule[7] += score;
    }
    else if (j == 0) {
        score = 50;
        rule[7] += score;
    }
}
```

#8-3 한쪽 or 양쪽 다 막혀 있는 경우)




```
//한쪽이 막혀 있다. !OO? ?는 not empty !는 상대방돌이 아님
//horizontal
if (i<5 && i>0 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1 &&
    (current_state->board[i - 1][j] == 0 || current_state->board[i + 2][j] == 0)) {
    if ((current_state->board[i - 1][j] == 0 && current_state->board[i + 2][j] == 0)) {// xoox 와 같이 양쪽 다 막힘.
        score = 5;
        rule[7] += score;
    }
    else { // 한쪽만 막힌 경우
        score = 10;
        rule[7] += score;
    }
    //printf("20\n");
}

if (i == 0 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1) { // left wall
    if (current_state->board[i + 2][j] == 0) { // 양쪽 다 막힘.
        score = 5;
        rule[7] += score;
    }
    else {
        score = 10;
        rule[7] += score;
    }
}

if (i == 5 && current_state->board[i][j] == 1 && current_state->board[i + 1][j] == 1) { // right wall
    if (current_state->board[i - 1][j] == 0) { // 양쪽 다 막힘.
        score = 5;
        rule[7] += score;
    }
    else {
        score = 10;
        rule[7] += score;
    }
}
```

돌을 착수하여 연속 2개의 돌을 만들 때 양 옆이 벽 또는 상대방의 돌로 막혀 있는 경우에 대한 점수를 부여하였다. 벽 또는 상대방의 돌로 한쪽만 막힌 경우는 10점을 부여하고, 벽 또는 상대방의 돌로 양쪽이 다 막힌 경우는 5점을 부여하였다. 양쪽이 막힌 경우에 양수의 점수를 부여하는 이유는, 초반의 경우에는 세로 또는 대각선의 시작의 가능성이 있기에 최소한의 양수 점수를 주었다.

#8-4 단순하게 하나의 column에 2개 연결되는 경우)

가로, 대각선에 비해 세로(vertical)은 생각해야할 조건이 비교적 쉬워 마지막 경우로 생각하였다. 세로에서 연속 두개로 쌓는 경우는, 당연하게 한쪽은 막힌 경우이기에 위에서 한쪽만 막혔을 때 부여했던 점수와 같은 10점은 주었다.

```
//column에만 두개 // vertical
if (j>0 && j < 5 && current_state->board[i][j] == 1 && current_state->board[i][j - 1] == 1 && current_state->board[i][j + 1] == C4_NONE) {
    score = 10;
    rule[7] += score;
}
```

<룰 정리 표>

Rule	설명	Score
AI Connect4	0000	5000000
Human Connect4 막기	_XXX0	250000
	0XXX0	300000
	XX0X / X0XX	250000
AI Connect3	_000_ both below full	10000
	_000_ XOR below full	1000
	_000_ both below empty	2000
	X000_, _000X, 벽000_, _000벽, below empty	1000
	X000_, _000X, 벽000_, _000벽, below full	300
	X000X, 벽000X, X000벽	-200
양 옆이 빈 Human Connect3 막기	_XX0_, _0XX_, _X0X_ both below full	5000
	_XX0_, _0XX_, _X0X_ XOR below full	150
AI / Human 7자 모양 만들기/막기	7, 좌우반전7, 상하반전7, 상하좌우반전 7	2000
	7, 좌우반전7, 상하반전7, 상하좌우반전 7 ,4개만들가능성 X	-100
두면 다음에 진다	Human 의 connect4를 도와주는 받침이 될 때	-10000
Connect2	00_0 ,0_00 below empty	1000
	00_0, 0_00 below full	300
	_00_ below both empty	150
	_00_ below XOR full	100
	_00_ below both full	50
	X00X, 벽00X, X00벽	5
	X00#,#00X, 벽00#, #00벽 (#은 상대방 돌이나 벽이 아닌 경우)	10
Connect1	00_0_00 both below full	10000
	00_0_00 both below empty, 00_0, 0_00 below empty	1000
	00_0_00 XOR below full	500
	00_0, 0_00 below full	300
	0_0_0 both below full, _0_0_ center below empty	200
	0_0_0 both below empty, _0_0_ center below full	150
	0_0_0 XOR below full	100

3. 시행착오

- JAVA 에서 C로 언어 변경을 하였다. 행과 열에 대한 정보를 저장하기에 포인터를 사용하는 것이 더 구현하기에 편했기 때문이다. 해당 JAVA 파일을 과제 소스코드와 함께 첨부하였다.

```
if (j < 3 && i < 4 && current_state->board[i][j] == 0 && current_state->board[i + 1][j + 1] == 0 && current_state->board[i + 2][j + 2] == 0 && current_state->board[i + 3][j + 3] == 1) {
    if (i > 0 && j > 0 && current_state->board[i - 1][j - 1] == 1) {
        score = 300000;
        rule[1] += score;
    }
    //      if (current_state->board[i][j-1] != C4_NONE && current_state->board[i + 4][j + 3] != C4_NONE) {
    score = 250000;
    rule[1] += score;
    //        printf("2-11\n");
    //        }
    |
    //diagonal _XXXO(_!=X) ascending
}
/*
//first line
if (j == 0 && j < 3 && i < 4 && current_state->board[i][j] == 0 && current_state->board[i + 1][j + 1] == 0 && current_state->board[i + 2][j + 2] == 0 && current_state->board[i + 3][j + 3] ==
score = 250000;
rule[1] += score;
//   printf("2-12\n");
//diagonal _XXXO(_!=X) ascending
```

- first 라인의 경우 받침이 있는 경우인데 받침 생각할 때 first line 고려하지 않았었다. 그래서 후에 first 라인을 새로 추가해주고 원래 있던 코드들과 합쳐주었다.
- 처음 코드를 짤 때, 룰이 아닌 휴리스틱에서 전체 보드판에 따라 점수를 부여하는 식으로 구현했었다. 이는 후에 룰을 구현하는 것에 큰 도움이 됐고 필요한 룰을 더 깊게 생각하는 계기가 되었다.
- 원래 (*7.내 차례 시에 나에게 유리하게 하도록 만든 룰이다. 내 차례 일 때 나의 돌이 2개 연결되어 있는 상황이라면 이를 이용해 나의 돌을 3개로 연결시킨다)* 라는 룰이 존재하였는데, 4개를 만드는 가능성만 고려하여 _ooo || ooo_ 이렇게 간단하게 조건을 설정하였었다. 그 뒤로, 수많은 실행 끝에 받침의 유무와 더 다양한 3개의 모양 꼴에 대해 고려해함을 깨닫고 현재 룰 3을 추가하였다. 위의 룰은 현재 코드에서 주석 처리 되어 있다.
- 또한 룰 3을 추가하면서, 상대방이 돌 3개를 두는 형태도 막는 룰 4도 같이 구현하였다.
- 7자를 만드는 rule 5에서 처음에는 7자 모양만 고려하였는데 실행 결과를 통해, 의미가 없는 7자도 있을 수 있음을 깨닫게 되었다. 4개 만들 가능성이 있을 때만 7자가 의미를 가지고 7자의 각 3들의 양끝이 이미 채워져있다면 무의미해지기에 그것에 대한 조건을 추가하였다.
- rule 8-1에서 first line을 따로 하지 않고 if 문 안에 넣어주어 중복되지 않게 수정하였다.
- rule 8-3에서 한쪽 막힌 경우에서 처음에는 돌로만 막힌 경우 생각, 벽도 고려해서 코드 수정하였다.

- 처음에 룰을 짤 때, AI가 1, 2 개를 두는 상황을 고려하지 않았다. 좀 더 크리티컬한 경우에만 룰을 만들어 실행하였더니, 초반에 룰에 기반하여 두지 않고, 오로지 처음에 컬럼에 부여된 가중치로만 실행하여 수정하였다.
- 룰 조건문을 else if -> if 로 바꾸었다. 처음에는 42개의 보드판을 다 보기에 각 룰마다 배제되어 읽어야 한다고 생각했다. 그러나, 하나의 위치에서 두개 이상의 룰이 적용되면 점수가 높아지거나 변동 사항이 생겨 다 if 로 변경하고 그에 따른 추가적 수정도 진행하였다.

4. 대결결과

휴리스틱 vs 룰 => 휴리스틱 WIN
휴리스틱 vs 퍼펙트 솔버 => 퍼펙트 솔버 WIN
룰 vs 퍼펙트 솔버 => 퍼펙트 솔버 WIN
휴리스틱+룰 vs 퍼펙트 솔버 => DRAW

5. 결론

휴리스틱을 직접 구현하려 했던 과정에서 수업시간에 배운 minmax 알고리즘과 이를 더 효율적으로 활용할 수 있는 negamax 를 알게 된 계기가 되었고, 개념에 대해 보다 심화적으로 생각할 수 있었다. 결국 오픈소스를 활용하기는 했지만 java를 통해 직접 코드를 짜보며 개념 복습과 이해에 있어 좋은 경험이 되었다. 또한 이 때 연구했던 heuristic은 좀더 치밀한 rule를 만드는데에 도움이 되었다.

초반의 시행착오였던 휴리스틱에 점수를 부여하는 방식을 룰에 가져오게 되면서 룰에 대해 많은 고민을 하여 구체적이고 세세한 부분까지 체크하는 룰을 구현하였다. 룰에 점수를 부여하는 과정에서 룰의 우선순위를 고민하면서 추가해야 할 룰과 겹치는 룰이 무엇인지 파악할 수 있었다. 또한 보드 전체를 확인하는 코드로 인해 발생한 룰 점수의 누적현상으로, 부여할 점수 선택에 있어 몇번의 시행착오와 연습게임을 거쳐 보다 안정적인 룰 적용 함수를 구현할 수 있었다.

우리 조는 인덱스 에러, 룰 논리 오류 문제 등 시행착오를 꽤나 많이 겪었고 그로 인해 조원들이 많이 힘들어 했지만 지나고 나서 보니 많은 시행착오를 했던 경험들이 축적 되어 connect4 게임과 min max 알고리즘, rule에 대해 깊은 이해를 가능하게 했다는 생각이 들었다. 뿐만 아니라 기본적으로 코딩을 함에 있어서 주의해야할 사항이나 확인해야할 부분이 무엇인지 알게 되는 프로젝트였다. 기본적으로 현재 상태에 대한 점수를 구한다는 점에서 heuristic기반 선택은 exploitation의 역할을 하였고 현재 주어진 상태에서 추정하여 점수를 부여한다는 점에서 rule 기반 선택은 exploration의 역할을 하였다. 수업 시간에 배운 search 전략의 두 가지 관점을 몸소 체험할 수 있는 프로젝트였다. 단순히 search algorithm을 구현하고 rule을 정의하는 것을 넘어 실제 경기에 있어 이 둘의 균형을 어떻게 해야 잘 유지할 수 있는지 연구하게 되었다. 실제로 우리의 heuristic과 rule은 하나로만 경기를 하면 perfect solver에게 매번 지기 일수였다. 하지만 rule과 heurisitc 두가지를 한 경기에 같이 이용하였을 때는 perfect solver와 비기는 결과까지 얻을 수 있었다. 이번 프로젝트에서 첫 수에는 가운데에 두면 안된다는 규칙이 없었다면 perfect solver와의 경기에서 승리할 수 있었을지도 모른다는 생각이 든다.

6. 보완할 점

*보드를 읽는 방식

컬럼을 0~6 부터 차례대로 for문을 돌리면서 drop_piece로 각 컬럼마다 채워져 있는 row를 계산하여 해당하는 위치에 7번 돌을 놓는다. 각 경우마다 돌을 놓은 곳을 놓았다고 가정하고 그 미래의 보드판에서 모든 위치인 42개의 칸을 읽어 룰을 판단한다. 이 방법이 처음에는 룰의 조건을 표현할 때 좀 더 용이하여 선택하였다.

42개의 모든 칸을 읽지 않는 다면 다음과 같이 룰을 처리해야했다. 예를 들어, oo_o 다음과 같은 모양을 판단해야 할 때, 내가 놓은 위치에서만 판단하려면, 놓은 위치를 기준으로 모든 방향과 모양을 달리 생각해야한다. 놓은 위치가 oo의 첫번째 일때는 옆은 내돌, 옆옆은 빈칸, 옆옆옆은 내돌 이다. 그러나 놓은 위치가 oo의 두번째 일때는 또 달라진다. 하지만, 내가 놓은 돌을 고정하고 그 보드판에 대하여 전체를 읽으면 oo_o 모양은 한번만 지정해주면 첫번째 위치일 때, 두번째 위치일 때 … 계속해서 검사하기 때문에 룰 조건을 표현하기에 좀 더 용이하였다.

하지만 룰을 다 구현하고 점수를 계산하는 과정에서, AI와 상대방이 번갈아가면서 턴을 반복할 때마다, 모든 돌이 놓여진 보드판을 불러와 문제가 발생하였다. 그 전 턴에서 계산하여 막았던 위치도 보드판에 놓여져 있으니 중복으로 룰을 검사하여 점수 산출에 조금 어려움을 겪었다. 이 문제점은 여러번 perfect solver와의 대결과, 우리조의 heuristic과의 대결, 사람과의 대결을 통해 점수 계산의 착오를 잡아내서 해당하는 점수를 부여하는 것으로 해결하였다.
처음부터 보드를 읽어 오는 방식이 좀 더 명확했다면 이런 오류가 나타나지 않았을 것이다.

7. 기여도 및 협업 과정

a. 기여도

아래의 협업과정을 보면 알 수 있듯이 3명의 조원 모두 이번 인공지능 과제에 적극적으로 참여하였다. 각각 파트를 나눠서 룰을 만들고 스카이프로 영상통화를 하며 함께 수정해나갔으며, 보고서 또한 구글 공유 문서로 함께 작성하였다.

우지수 :

코드 – rule 5, 6, 8, rule 3 베이스 작성(각주 처리 된 rule 7), 메인함수 수정(sprintf & get_num), 인덱스 및 논리 오류 수정

보고서 – 전반적인 개요 작성, polmaki 코드 분석 후 heuristic 및 negamax 설명 작성, 룰 함수 5, 6, 8 설명, 보완할 점, 협업과정, 시행착오 작성, 마지막 워드문서로 통합

사전준비 – 보고서 공유 문서 생성, 룰 점수 배분, connect4로 글씨 'AI' 만들기!, 룰 공유 문서 작성, rule 자료 조사 및 편집

전지연 :

코드 – java heuristic 구현, rule 4, 7 구현, rule3 일부 구현, 인덱스 및 논리 오류 수정, rule 적용 함수(apply_rule, eval_rule)알고리즘 제시, 틀 구현, 메인함수 수정(룰 모드 추가), 착수점 프린트 구현

보고서- 알고리즘의 개념 및 배경적 지식(게임트리, 알파베타프루닝, 민맥스), 룰 함수 전체적인 설명, 룰 1,3 설명, 결론 작성, 시행착오 작성, 대결결과, 기여도 작성

사전준비- goorm 컨테이너 생성, 룰 공유 문서 작성, rule 점수 표 생성, 룰 점수 배분, rule 자료 조사 및 편집

정예주 :

코드 - rule 1, 2, 3, 4 구현, 인덱스 및 논리 오류 수정

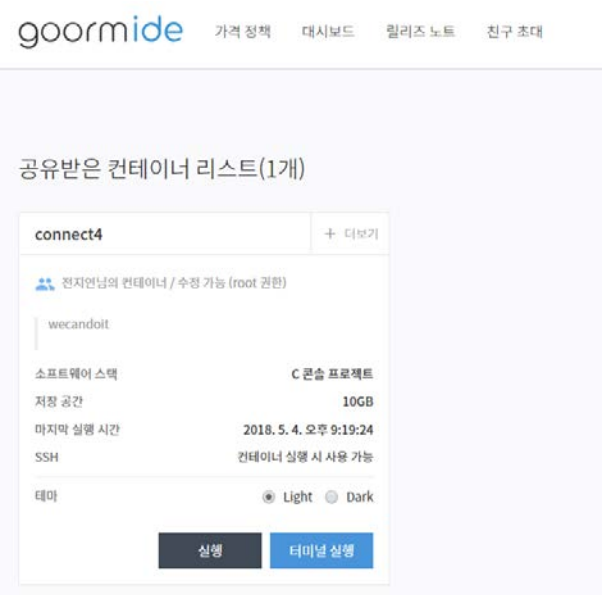
보고서 - rule 2, 4, 7 설명, heuristic 설명과 결론 작성

사전준비 - 룰 공유 문서 생성 및 작성, rule 자료 조사 및 편집, 실행되는 화면 영상 촬영 및 편집, 룰점수 배분

이 시대의 최고의 타이퍼 - 맥 최고, 중간에 코드의 시스템 오류로 윈도우 기반 노트북에서 실행창이 강제 종료되는 에러가 있어, 맥을 사용하는 정예주 학우가 스카이프로 화면 공유를 할 때, 실행창을 돌리고 수정하는 코드를 작성하는 고된 작업을 시행

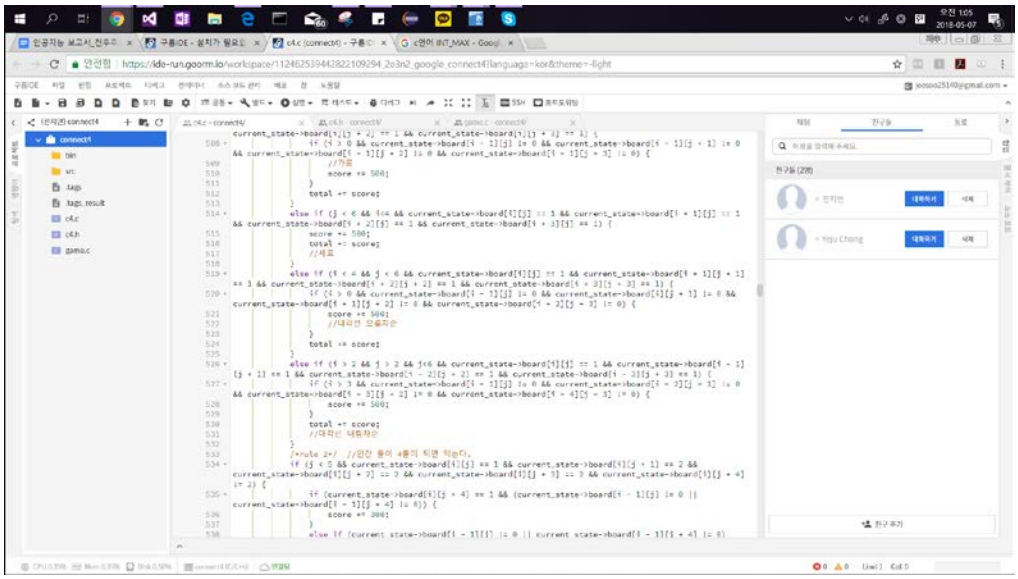
b. 협업과정

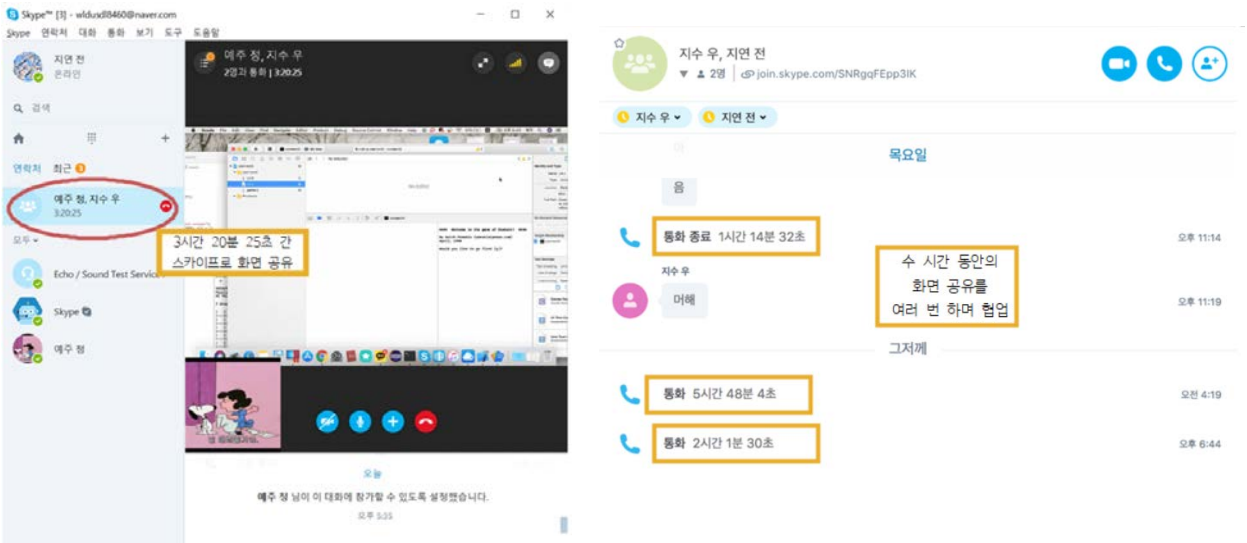
처음에 우리 조가 구현할 인공지능에서 어떤 heuristic과 rule을 사용할 것인지 의논하기 위해, 구글 공유 문서를 이용하여 함께 작성하고 수정된 사안을 모두가 바로바로 확인할 수 있도록 하였다.



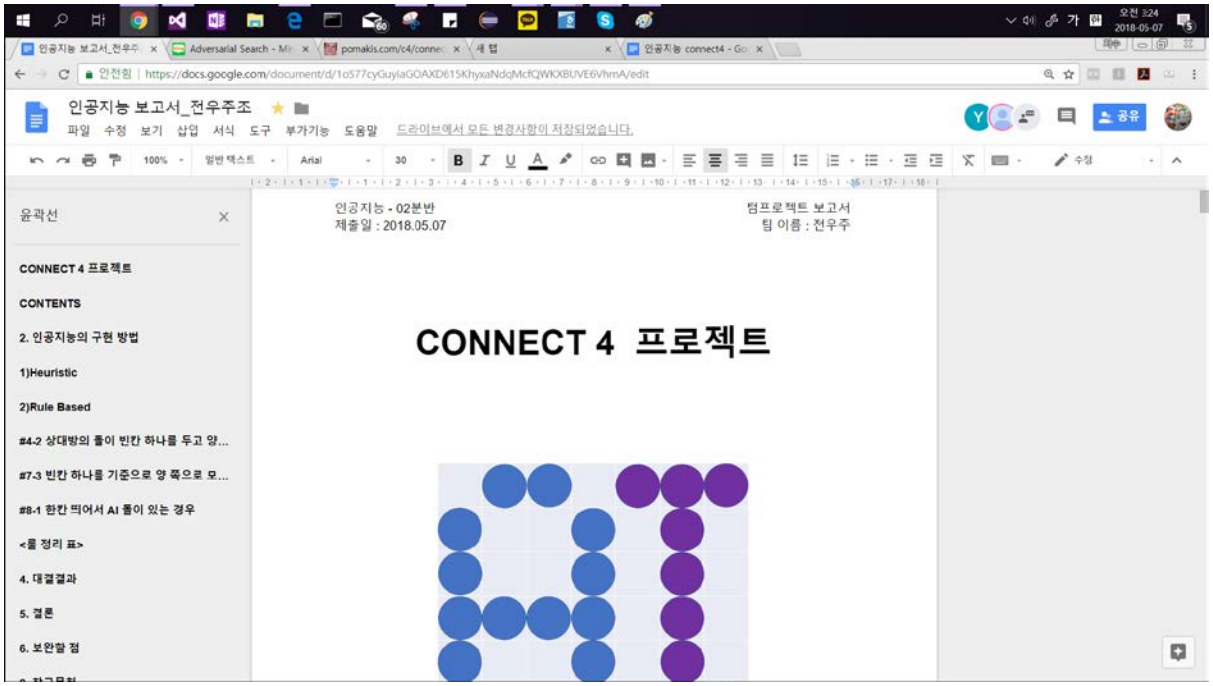
동시에 코드를 작성하고 채팅을 통해 의논할 수 있는 goorm.ide를 이용하였다. 원래는 git-hub 기반인 cloud9을 이용하려 하였으나, 로그인 문제와 클라우드가 작동하지 않아 좀 더 간편한 goorm.ide를 사용하였다. 컨테이너를 만들고 공유하는 과정은 어느 사이트보다 간편하였지만 서로 저장하고, 코드를 수정하면 잘 보이지 않거나, 서버가 연결이 잘 안되는 등의 어려움이 있었다. 그래서 goorm.ide 는 전체적인 틀을 작성할 때 주로 사용하였다.

goorm.ide 의 실행 화면 및 채팅과 조원들의 목록을 볼 수 있다.

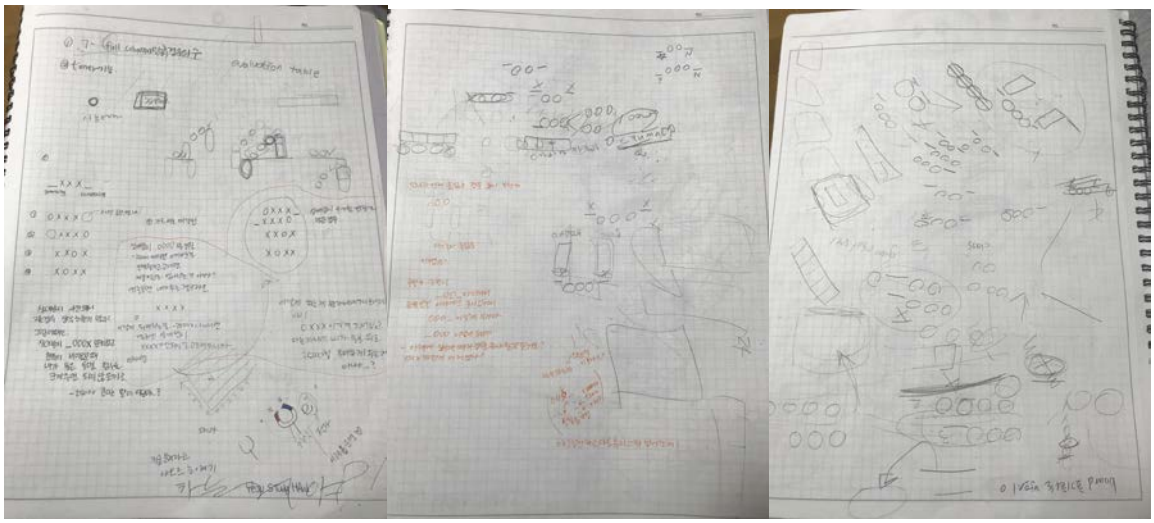




코드를 실행해보면서, 룰의 수많은 조건들의 인덱스 오류나 실행 결과를 통해 시행착오를 겪으며 수정하는 과정은 평일에는 학교에서 다같이 정보관에서 모이는 시간을 가졌다. 다 함께 모이지 못하는 경우에, 스카이프 화면 공유를 통하여, 같이 실행을 하고 그 결과에 대해 분석하여 미흡했던 룰을 여러 번 수정하는 과정을 거쳤다. 룰을 구현하는 과정에서는 각자 분담하여 짜서 코드를 합쳤는데, 그 과정에서 보드나 인덱스에 대한 서로의 개념이 달라서 발생한 오류는 다같이 수정하였고 서로가 이해하고 온전히 텀프로젝트에 다 참여할 수 있도록 배려하였다.



보고서를 작성하는 과정도 다음과 같이 구글 공유 문서를 이용하여 작성하였다. 처음에는 분담하여 보고서를 작성한 후, 부족한 부분을 보완하고 전반적인 코드 구현에 대한 설명과 결론 및 보완할 점 등 은 다 같이 의논 하에 작성하였다.



초반 휴리스틱과 룰의 조건을 구현할 때는 직접 만나 다음과 같이 그려주고 설명해주며, 다같이 작업하였습니다.

8. 참고문헌

http://pomakis.com/c4/connect_generic/

http://blog.gamesolver.org/ //perfect solver