COSE361 – 02

인공지능 Term Project-Connect 4 보고서

바둑이와 애용이

2017320206 조동혁, 2017320187 고도환

1. Basic Idea - Search

기본적으로 인공지능 수업 시간에 배운 내용 중 Minimax algorithm을 사용하여 구현하였다. Alpha Beta pruning을 사용하지 않은 것은 단말 노드까지 모든 노드를 펼칠 수 있도록 코드를 구현하였고, 이때 Minimax 알고리즘 방식만을 사용하더라도 각 착수에 매우 적은 시간이 소요되기 때문에 굳이 Alpha Beta pruning을 사용할 필요성을 느끼지 못하였기 때문이다.

6x7보드에서 게임이 진행되므로, 특정 시점에 돌들이 놓여져 있는 것을 하나의 state로 본다면, 바로 다음 state는 각기 다른 열에 착수 하는 7가지 경우가 존재한다. 이를 tree에서의 부모 노드 및 자식 노드의 개념과 결합하여, 현재 상태에서 7수 앞까지 내다볼 수 있는, 즉 depth가 7인 search tree를 구성하였다. 단말 노드에서부터 Minimax algorithm을 통해 value값을 루트 노드까지 가지고 올 수 있도록 했다. Depth가 7이고 모든 단말 노드를 다 확인해야 하기 때문에 단순 반복문으로 구현하기에는 가독성이 떨어지고 7중 포문이 필요해 복잡해질 것이라고 생각했고 고민 끝에 포인터와 재귀함수를 사용하여 모든 노드가 펼쳐지도록 구현하였다.

처음에 실제로 값을 갖는 것은 단말 노드들 뿐이고, 재귀 함수를 통과하면서 자식 노드들의 값들 중 min또는 max값이 단계에 맞게 부모 노드의 값으로 계속 해서 update될 수 있도록 했다. 따라서 단말 노드들의 값 들 중 하나가 루트 노드로, 일부가 루트 노드의 자식 노드들의 값으로 업데이트 된다. 모든 과정을 수행하고 나면 main함수에서 7개의 자식 노드 중에 일치하는 값을 가지는 노드를 선택하도록 했다. Minimax에서 max연산의 역연산을 해주어 결과적으로 올바른 단말 노드로 나아갈 수 있도록 구현한 것이고, 이를 통해 인공지능이 최적의 수를 다음 수로 착수할 수 있도록 하였다. 이 과정을 AI의 차례가 될 때 마다 그 때의 게임 보드의 상태를 루트 노드로 하여 반복하게 된다.

코드의 최초 버전에서는 overflow, 즉 해당 열에 더 이상 빈칸이 없다는 것을 보여주기 위해 넣어둔 값인 -10000이 Minimax에서 걸러지지 않고 루트 노드의 값까지 올라오는 문제가 있었다. 어느 한 column에 overflow가 발생하면 Minimax algorithm이 올바르게 작동하지 못하고 특정 column만을 계속해서 선택하는 상황이 발생했다. 이를 해결하기 위해 모든 자식 노드의 값이 -10000이면 -10000이 부모 노드의 값으로 올라가지만, 하나라도 -10000이 아닌 값이 있으면 -10000을 아예 비교에서 제외한 채 max또는 min 값을 찾는 방

법을 채택하였다. 이를 통해 어느 한 column에 overflow가 발생하더라도 올바르게 다음에 착수할 열을 정할 수 있었고, 성능의 향상을 이루어 낼 수 있었다.

아래는 해당 부분에 해당하는 코드와 그에 대한 주석이다.

```
search_tree *fmax(search_tree *pre) {
//minimax 알고리즘에서 max부분 --아이디어 회의: 고도환(*),조동혁 코드작성: 고도환,조동혁 주석: 조동혁
        int max = pre->child[0]->value; //max값을 가장 왼쪽 자식노드의 value값으로 초기화
        int count = 0;                  //overflow가 발생한 경우 비교에서 제외하기 위해
                                         overflow가 발생한 노드의 개수 counter.
        search_tree *temp = pre->child[0]; //반환할 노드를 가장 왼쪽 자식 노드로 초기화

        for (int i = 0; i < c_num; i++) {  //overflow가 발생한 자식 노드의 개수 count해줌
                if (pre->child[i]->value == -10000) count++;
        }

        if (count != 7) {                   //하나라도 overflow가 발생하지 않는 노드가 있는 경우
        for (int i = 0; i < c_num; i++) {
                                            //overflow가 발생하지 않은 최초의 자식노드를 찾아
                                            max값과 temp값을 다시 초기화해줌.
                if (pre->child[i]->value != -10000) {
                        max = pre->child[i]->value;
                        temp = pre->child[i];
                        break;
                        }
                }

          for (int i = 0; i < c_num; i++){          //다시 초기화 한 max값을 바탕으로
                                                    overflow가 발생하지 않는 자식 노드들 중 가장 큰
                                                    value값을 가지고 있는 노드를 찾아 temp에 저장.
                if (max <= pre->child[i]->value && pre->child[i]->value != -10000){
                max = pre->child[i]->value;
                temp = pre->child[i];
                        }
                }
        }
        return temp;                            //가장 큰 value값을 가지는 노드 반환
}
search_tree *fmin(search_tree *pre) {
//minimax 알고리즘에서 min부분 --아이디어 회의: 고도환,조동혁 코드작성: 고도환(*),조동혁 주석: 조동혁
        int min = pre->child[0]->value;      //min값을 가장 왼쪽 자식노드의 value값으로 초기화
        int count = 0;                       //overflow가 발생한 경우 비교에서 제외하기위해
                                              overflow가 발생한 노드의 개수 counter.
        search_tree *temp = pre->child[0];    //반환할 노드를 가장 왼쪽 자식 노드로 초기화

        for (int i = 0; i < c_num; i++){      //overflow가 발생한 자식 노드의 개수 count해줌.
                if (pre->child[i]->value == -10000) count++;
        }

        if (count != 7){                    //하나라도 overflow가 발생하지 않는 노드가 있는 경우
                for (int i = 0; i < c_num; i++) {   //overflow가 발생하지 않는 최초의
                                                    자식노드를 찾아 min값과 temp값을 다시
```

```
                                                초기화해줌.

                                if (pre->child[i]->value != -10000) {
                                min = pre->child[i]->value;
                                temp = pre->child[i];
                                break;
                        }
                }

                 for (int i = 0; i < c_num; i++) {       //다시 초기화 한 min값을 바탕으로
                                                         overflow가 발생하지 않는 자식 노드들
                                                         중 가장 작은 value값을 가지고 있는
                                                         노드를 찾아 temp에 저장
                 .
                        if (min >= pre->child[i]->value && pre->child[i]->value != -10000) {
                                min = pre->child[i]->value;
                                temp = pre->child[i];
                        }
                }
        }

        return temp;                    //가장 작은 value값을 가지는 노드 반환
}

void find_path(search_tree *pre, int num) {
//minimax알고리즘을 fmin함수와 fmax함수를 통해 구현한 부분 --아이디어 회의 고도환,조동혁
코드작성: 고도환(*),조동혁. 주석: 조동혁

        int i;
        int temp = num;

        if (pre->child[0] == NULL)      //단말 노드까지 내려온 경우 재귀 호출 중단하고 종료.
                return;
        else {
                temp--;         //단말 노드까지 도달하지 못한 경우 한 단계 낮은 depth로 확장.
                for (i = 0; i < c_num; i++)
                        find_path(pre->child[i], temp);
        }

        if (num % 2 == 0 {                  //짝수 depth의 경우 min함수 사용하여 자식 노드 중
                                            가장 작은 value의 값을 부모 노드의 value로 저장
                pre->value = fmin(pre)->value;

        }
        else if (num % 2 == 1) {        //홀수 depth의 경우 max함수 사용하여 자식노드 중
                                        가장 큰 value의 값을 부모노드의 value로 저장
                pre->value = fmax(pre)->value;
        }
}
```

2. Heuristic Function

휴리스틱 함수는 가장 많은 시행착오를 겪은 부분이었다. 독창성과 성능을 모두 잡아야 했기 때문에, 어떤 식으로 휴리스틱을 구성해야 할 지에 대해 생각하는 것이 이번 프로젝트에서 가장 많은 시간을 소요한 부분인 것 같다. 처음에 생각했던 것은 전체 보드를 가능한 모든 연결된 4칸의 조합으로 나누어서 살펴보는 것이었는데, everycoding 사이트를 살펴보니 그동안 가장 흔했던 휴리스틱이었기 때문에 좀 더 독창적으로 함수를 구현하고자 채택하지 않았다. 이 방법 외에도 보드를 여러 개의 3x3 보드로 나누어서 보는 방법도 생각 해봤지만 사고실험 결과 실용적이지 않다고 판단되어 진행하지 않았고 현재 상태에서 돌이 놓인 column에서 인접한 column에 높은 휴리스틱 값을 주는 방법은 실제로 사용해 보았지만 그 성능이 현저히 떨어져 중간에 폐기하였다. 다음에 도입한 것은 winning line개념이었다. 이 역시 everycoding 사이트에서 본 표를 바탕으로 아이디어를 얻었다.(Figure 1)

각 칸마다 이길 수 있는 경우의 수가 정해져 있기 때문에 이를 활용하여 각 칸마다 휴리스틱 점수를 부여하고자 했다. 초기 버전에서는 단순히 winning line만 가지고 다음 수를 착수했기 때문에 greedy algorithm의 형태를 벗어나지 못했고, 자연히 성능도 이전 휴리스틱에 비해 발전하기는 했지만 그다지 좋지 못했다.

이를 보완하기 위해, 가중치 개념을 도입했다. 7수 앞까지 내다본, 즉 단말 노드 상태에서 자신의 돌이 2개 이상 연결되어 있으면 가중치를 곱해서 값을 계산하도록 하였다. 앞서 언급했던 winning line을 게임 보드와 같은 크기의 배열 winning_line[6][7]로 선언하여 각 칸별로 이길 수 있는 경우의 수를 배열에 넣어주었다. 점수는 연결되어 있는 돌들의 위치의 winning_line배열 값의 합과 가중치를 곱하는 방식으로 계산하였다.

이 때 가중치는 돌이 2개 연결되어 있으면 10, 3개 연결되어 있으면 20, 4개 연결되어 있으면 100을 주었다. 가중치를 어떻게 설정할지에 대해서도 많은 고민을 하였는데, 가운데 column일수록 winning line값이 크기 때문에, 가운데에서 먼 column에서 2개의 돌이 연결된 경우에 단순히 가운데에 한 개의 돌이 있는 경우 보다 큰 value를 가질 수 있도록 해야 했다. 가장 큰 winning line value가 13이기 때문에 그 칸의 winning value와 가중치를 곱했을 때 13보다 크도록 하기 위해 13과 가까운 값인 10을 돌 2개의 가중치로 선택했다. 돌 3개와 돌 4개일 때의 가중치는 비슷한 관점에서 출발하여 계산을 통해 근사값을 산출한 뒤, 여러 번의 실험을 통해 가장 성능이 좋은 값으로 조정하였다. 그 결과 돌 3개의 가중치는 20, 돌 4개의 가중치는 100일 때 만족할 만한 성과를 보았기 때문에 이 값들을 가중치로 정하였다.

AI의 차례가 되면 각 단말 노드에서의 게임 보드의 상태가 주어지고, value를 0으로 초기화한다. 자신의 돌이 여러 개 연결되는 경우에는 winning line값과 가중치와 곱한 값을 점수에 더해주고, 상대의 돌이 여러 개 연결되는 경우에는 winning line값과 가중치를 곱한 값을 반대로 점수에서 빼 준다. 즉 어떤 column에 대한 휴리스틱 값 H(n)은 다음과 같이 계산된다.

H(n) = E(m)-E(o) : E(m)->내가 승리하는 경우에 대한 평가 값, E(o) -> 상대가 승리하는 경우에 대한 평가 값.

이 때, 내가 승리하는 경우를 계산하는 데 사용하는 가중치와, 상대가 승리하는 경우를 계산하는데 사용한 가중치가 다르면 문제가 발생한다. 자신의 승리에 대한 가중치가 높으면 막아야 하는 상황에서도 이를 고려하지 않고 자신이 이길 가능성이 높은 열에 착수하게 되고, 반대의 상황에서는 이길 수 있음에도 상대의 승리를 우선적으로 막으려 한다. 따라서 두 경우의 가중치를 똑같이 주어 공격과 수비 한 쪽에 우선순위를 두지 않고 이상적인 평가를 내릴 수 있도록 하였다.

아래는 구체적인 휴리스틱 함수와 그에 대한 주석이다.

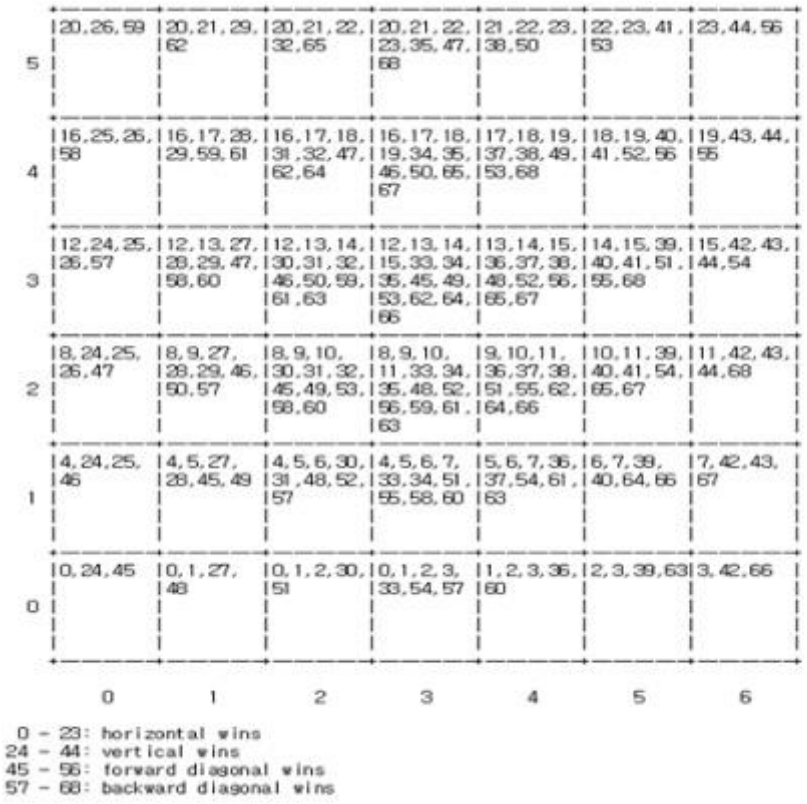

Figure 1

```
int heuristic_function(int arr[][7], int turn) {
//노드에 대한 휴리스틱 값을 찾아주는 함수이다. --아이디어 회의: 고도환 조동혁, 코드 작성: 고도환 주석: 조동혁

int winning_line[6][7] = { { 3, 4, 5, 7, 4, 3 },{ 4, 6, 8, 10, 8, 6, 4 },{ 5, 8, 11, 13, 11, 8, 5 },{ 5, 8, 11,
13, 11, 8, 5 },{ 4, 6, 8, 10, 8, 6, 4 },{ 3, 4, 5, 7, 4, 3 } };

//보드의 각 칸 별로 이길 수 있는 경우의 수를 배열에 저장하여 가중치와 곱해 휴리스틱 값 계산에 사용한다.

int i, j;                                   //for문에 사용될 변수들
int value = 0;                                  //value값 초기화
```

```
const int m2 = 10;
 //7수 앞을 내다본 보드 상태에서 자신의 돌이 2개 이어져 있는 경우의 휴리스틱 가중치.
const int m3 = 20;
//7수 앞을 내다본 보드 상태에서 자신의 돌이 3개 이어져 있는 경우의 휴리스틱 가중치.
const int m4 = 100;
//7수 앞을 내다본 보드 상태에서 자신의 돌이 4개 이어져 있는 경우의 휴리스틱 가중치.
const int y2 = 10;
//7수 앞을 내다본 보드 상태에서 상대의 돌이 2개 이어져 있는 경우의 휴리스틱 가중치.
const int y3 = 20;
//7수 앞을 내다본 보드 상태에서 상대의 돌이 3개 이어져 있는 경우의 휴리스틱 가중치.
const int y4 = 100;
//7수 앞을 내다본 보드 상태에서 상대의 돌이 4개 이어져 있는 경우의 휴리스틱 가중치.

for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 자신의 돌이 2개 이어져 있는 경우에 value값에 가중치 부여.
          for (j = 0; j < 6; j++) {
                    if (arr[i][j] == arr[i][j + 1] && arr[i][j] == turn) {
                                        value += (winning_line[i][j] + winning_line[i][j + 1]) * m2;
                                                  j += 2;
                                        }
                              }
                    }

for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 자신의 돌이 3개 이어져 있는 경우에 value값에 가중치 부여.
          for (j = 0; j < 5; j++) {
                    if (arr[i][j] == arr[i][j + 1] && arr[i][j] == arr[i][j + 2] && arr[i][j] == turn) {
                     value += (winning_line[i][j] + winning_line[i][j + 1] + winning_line[i][j + 2]) * m3;
                    j += 3;
                              }
                    }
          }

for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 자신의 돌이 4개 이어져 있는 경우에 value값에 가중치 부여.
          for (j = 0; j < 4; j++) {
                              if (arr[i][j] == arr[i][j + 1] && arr[i][j] == arr[i][j + 2] && arr[i][j] ==
                              arr[i][j + 3] && arr[i][j] == turn) {
                                        value += (winning_line[i][j] + winning_line[i][j + 1] +
                                        winning_line[i][j + 2] + winning_line[i][j + 3]) * m4;
                              }
                    }
          }


for (j = 0; j < 7; j++) {
//7수 앞을 내다본 보드 상태에서 세로 방향으로 자신의 돌이 2개 이어져 있는 경우에 value값에 가중치 부여.
                    for (i = 0; i < 5; i++) {
                              if (arr[i][j] == arr[i + 1][j] && arr[i][j] == turn) {
                                        value += (winning_line[i][j] + winning_line[i + 1][j]) * m2;
                                        i += 2;
                              }
                    }
          }

for (j = 0; j < 7; j++) {
//7수 앞을 내다본 보드 상태에서 세로 방향으로 자신의 돌이 3개 이어져 있는 경우에 value값에 가중치 부여.
          for (i = 0; i < 3; i++) {
                    if (arr[i][j] == arr[i + 1][j] && arr[i][j] == arr[i + 2][j] && arr[i][j] == turn) {
                       value += (winning_line[i][j] + winning_line[i + 1][j] + winning_line[i + 2][j]) * m3;
                              }
                    }
          }

for (j = 0; j < 7; j++) {
```

```
//7수 앞을 내다본 보드 상태에서 세로 방향으로 자신의 돌이 4개 이어져 있는 경우에 value값에 가중치 부여.
		for (i = 0; i < 2; i++) {
			if (arr[i][j] == arr[i + 1][j] && arr[i][j] == arr[i + 2][j] && arr[i][j] == arr[i + 3][j]
			&& arr[i][j] == turn) {
				value += (winning_line[i][j] + winning_line[i + 1][j] + winning_line[i + 2][j] +
				winning_line[i + 3][j]) * m4;
				}
			}
		}



for (i = 0; i <= 4; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 자신의 돌이 2개 이어져 있는 경우에 value값에 가중치
부여.
		for (j = 0; j <= 5; j++) {
			if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == turn) {
				value += (winning_line[i][j] + winning_line[i + 1][j + 1]) * m2;
				i += 2;
				j += 2;
				}
			}
		}

for (i = 0; i <= 3; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 자신의 돌이 3개 이어져 있는 경우에 value값에 가중치
부여.
		for (j = 0; j <= 4; j++) {
			if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == arr[i + 2][j + 2] && arr[i][j] == turn)
				value += (winning_line[i][j] + winning_line[i + 1][j + 1] + winning_line[i + 2][j
				+2]) * m3;
				}
			}
		}

for (i = 0; i <= 2; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 자신의 돌이 4개 이어져 있는 경우에 value값에 가중치
부여.
		for (j = 0; j <= 3; j++) {
			if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == arr[i + 2][j + 2] && arr[i][j] == arr[i +
			3][j + 3] && arr[i][j] == turn) {
				value += (winning_line[i][j] + winning_line[i + 1][j + 1] + winning_line[i + 2][j
				+ 2] + winning_line[i + 3][j + 3]) * m4;
				}
			}
		}


for (i = 5; i >= 1; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 자신의 돌이 2개 이어져 있는 경우에 value값에 가중치
부여.
		for (j = 0; j <= 5; j++) {
			if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == turn) {
				value += (winning_line[i][j] + winning_line[i - 1][j + 1]) * m2;
				i -= 2;
				j += 2;
				}
			}
		}

for (i = 5; i >= 2; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 자신의 돌이 3개 이어져 있는 경우에 value값에 가중치
부여.
		for (j = 0; j <= 4; j++) {
			if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == arr[i - 2][j + 2] && arr[i][j] == turn) {
				value += (winning_line[i][j] + winning_line[i - 1][j + 1] + winning_line[i - 2][j
```

```
                        + 2]) * m3;
                        }
                }
        }

for (i = 5; i >= 3; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 자신의 돌이 4개 이어져 있는 경우에 value값에 가중치
부여.
        for (j = 0; j <= 3; j++) {
                if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == arr[i - 2][j + 2] && arr[i][j] == arr[i -
                3][j + 3] && arr[i][j] == turn) {
                        value += (winning_line[i][j] + winning_line[i - 1][j + 1] + winning_line[i - 2][j
                        + 2] + winning_line[i - 3][j + 3]) * m4;
                        }
                }
        }

turn = 3 - turn;

//상대가 이기는 경우에 대해 가중치를 제거해 주기 위해 turn을 자신에서 상대로 바꾸어 준다.



for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 상대의 돌이 2개 이어져 있는 경우에 value값에서 가중치 빼준다.
        for (j = 0; j < 6; j++) {
                if (arr[i][j] == arr[i][j + 1] && arr[i][j] == turn) {
                        value -= (winning_line[i][j] + winning_line[i][j + 1]) * y2;
                        j += 2;
                        }
                }
        }

for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 상대의 돌이 3개 이어져 있는 경우에 value값에서 가중치 빼준다.
        for (j = 0; j < 5; j++) {
                if (arr[i][j] == arr[i][j + 1] && arr[i][j] == arr[i][j + 2] && arr[i][j] == turn) {
                    value -= (winning_line[i][j] + winning_line[i][j + 1] + winning_line[i][j + 2]) * y3;
                    j += 3;
                        }
                }
        }

for (i = 0; i < 6; i++) {
//7수 앞을 내다본 보드 상태에서 가로 방향으로 상대의 돌이 4개 이어져 있는 경우에 value값에서 가중치 빼준다.
        for (j = 0; j < 4; j++) {
                if (arr[i][j] == arr[i][j + 1] && arr[i][j] == arr[i][j + 2] && arr[i][j] == arr[i][j + 3]
                && arr[i][j] == turn) {
                        value -= (winning_line[i][j] + winning_line[i][j + 1] + winning_line[i][j + 2] +
                        winning_line[i][j + 3]) * y4;
                        }
                }
        }


for (j = 0; j < 7; j++) {
//7수 앞을 내다본 보드 상태에서 세로 방향으로 상대의 돌이 2개 이어져 있는 경우에 value값에서 가중치 빼준다.
        for (i = 0; i < 5; i++) {
                if (arr[i][j] == arr[i + 1][j] && arr[i][j] == turn) {
                        value -= (winning_line[i][j] + winning_line[i + 1][j]) * y2;
                        i += 2;
                        }
                }
        }

for (j = 0; j < 7; j++) {
```

```
//7수 앞을 내다본 보드 상태에서 세로 방향으로 상대의 돌이 3개 이어져 있는 경우에 value값에서 가중치 빼준다.
          for (i = 0; i < 3; i++) {
                    if (arr[i][j] == arr[i + 1][j] && arr[i][j] == arr[i + 2][j] && arr[i][j] == turn) {
                        value -= (winning_line[i][j] + winning_line[i + 1][j] + winning_line[i + 2][j]) * y3;
                              }
                    }
          }

for (j = 0; j < 7; j++) {
//7수 앞을 내다본 보드 상태에서 세로 방향으로 상대의 돌이 4개 이어져 있는 경우에 value값에서 가중치 빼준다.
          for (i = 0; i < 2; i++) {
                    if (arr[i][j] == arr[i + 1][j] && arr[i][j] == arr[i + 2][j] && arr[i][j] == arr[i + 3][j]
                    && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i + 1][j] + winning_line[i + 2][j] +
                              winning_line[i + 3][j]) * y4;
                              }
                    }
          }

for (i = 0; i <= 4; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 상대의 돌이 2개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 5; j++) {
                    if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i + 1][j + 1]) * y2;
                              i += 2;
                              j += 2;
                              }
                    }
          }

for (i = 0; i <= 3; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 상대의 돌이 3개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 4; j++) {
                    if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == arr[i + 2][j + 2] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i + 1][j + 1] + winning_line[i + 2][j
                              + 2]) * y3;
                              }
                    }
          }

for (i = 0; i <= 2; i++) {
//7수 앞을 내다본 보드 상태에서 대각선(좌상->우하) 방향으로 상대의 돌이 4개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 3; j++) {
                    if (arr[i][j] == arr[i + 1][j + 1] && arr[i][j] == arr[i + 2][j + 2] && arr[i][j] == arr[i +
                    3][j + 3] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i + 1][j + 1] + winning_line[i + 2][j
                              + 2] + winning_line[i + 3][j + 3]) * y4;
                              }
                    }
          }

for (i = 5; i >= 1; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 상대의 돌이 2개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 5; j++) {
                    if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i - 1][j + 1]) * y2;
                              i -= 2;
                              j += 2;
                              }
                    }
          }
```

```
for (i = 5; i >= 2; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 상대의 돌이 3개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 4; j++) {
                    if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == arr[i - 2][j + 2] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i - 1][j + 1] + winning_line[i - 2][j
                              + 2]) * y3;
                              }
                    }
          }

for (i = 5; i >= 3; i--) {
//7수 앞을 내다본 보드 상태에서 대각선(좌하->우상) 방향으로 상대의 돌이 4개 이어져 있는 경우에 value값에서
가중치 빼준다.
          for (j = 0; j <= 3; j++) {
                    if (arr[i][j] == arr[i - 1][j + 1] && arr[i][j] == arr[i - 2][j + 2] && arr[i][j] == arr[i -
                    3][j + 3] && arr[i][j] == turn) {
                              value -= (winning_line[i][j] + winning_line[i - 1][j + 1] + winning_line[i - 2][j
                              + 2] + winning_line[i - 3][j + 3]) * y4;
                              }
                    }
          }

return value;
//계산한 value값 반환해준다.
}
```

3. Rule – Based

휴리스틱 함수를 구현한 뒤, 대부분의 경우에 타 조의 AI와의 대결에서도 개발진인 우리와의 대결에서도 꽤나 뛰어난 성능을 보여주었으나, 1가지 경우에서 심각한 문제가 발견되었다. 상대의 돌이 휴리스틱이 낮은 가장자리 column에 3개 연결되어 있는 경우 막지 않는다는 것이었다. 고정적으로 7수 앞을 내다보고 그 상황에서 가장 좋은 수가 될 수 있게 결정하도록 휴리스틱을 구현하여서 막상 단 한 수 앞을 내다보아야 하는 경우에 그러지 못하는 경우가 발생한 것이다.

처음에는 휴리스틱 함수에서 상대의 돌이 3개 연결되어 있는 경우의 가중치를 자신의 돌이 3개 연결되어 있는 경우보다 높게 주어서 상대의 돌이 3개 연결되어 있으면 우선적으로 막도록 하려했으나 그 경우 끝낼 수 있음에도 상대의 돌을 우선적으로 먼저 막게 되어 경기를 끝내지 못하는 경우가 발생하게 되었다. 따라서 자신과 상대의 휴리스틱 가중치를 다르게 설정하는 것은 옳지 않다고 판단하고 이런 경우를 처리할 수 있도록 하는 rule의 필요성을 느껴 한 가지 rule을 만들게 되었다.

```
 int next_check(int arr[][7], int temp_arr[][7], int turn) {
//hueristic함수를 보완하기 위해 rule-base방식을 도입함. 상대의 돌이 3개 연결되어 있으면 바로 막고, 자신의 돌이 3개 연결되어 있으면 휴리스틱 값과 상관 없이 돌을 그곳에 둘 수 있도록 함. --아이디어 회의 고도환, 조동혁 코드 구현 고도환 주석 조동혁

        set_temp_arr(arr, temp_arr);
        //한 수 앞을 내다보기 위한 임시 게임보드를 만듦.
        for (int i = 0; i < c_num; i++) {
        //한 수를 두었을 때, 게임이 끝나면 해당 column을 반환.
                put(temp_arr, i, turn);
                if (check(temp_arr) != 0) return i;
        }
                return -1;
        //그렇지 않으면 -1을 반환
}
```

바로 다음 state7개만 확인하였을 때, 자신 또는 상대가 이기는 경우가 발생하면 그 column에 우선적으로 착수하도록 next_check라는 rule을 만들게 되었다. main함수에서 해당 상황이 발생하면 flag값이 바뀌어 휴리스틱 함수로 선택된 column보다 해당 column이 더 우선적으로 선택되도록 하였다. 또한 게임을 이기는 것이 목적이기 때문에 자신이 끝낼 수 있는 경우를 더 우선적으로 확인하도록 하였다.

결과적으로 최초에 발생했던 문제인 가장자리 열에 상대의 돌 3개가 연속적으로 놓여있을 때 막지 않는 것과 이를 수정하기 위해 휴리스틱 가중치 값을 변경해서 나타났던 공격보다 수비를 우선시하게 되는 문제를 모두 수정할 수 있었다. 흥미로운 사실은 rule과 해당하는 main함수를 고친 것 이외에는 코드를 손보지 않았는데 AI가 이전보다 흔히 쌍삼이라고 하는 3개가 연결된 돌 2개를 만드는 경우가 크게 증가했는데 이로 인해 사람이 느끼기에 훨씬 까다로워져 성능이 크게 향상되었다고 느껴졌다.

4. Participation

대부분의 코딩 작업은 고도환 학생이 리드하여 진행하였다. 어려운 부분, 예를 들어 휴리스틱 평가 함수나 Minimax algorithm을 구현하는 과정, rule 만들기는 같이 아이디어 회의를 한 후 같이 코딩을 하였다. 게임이 끝났는 지를 확인하는 check부분과 main의 부분은 고도환 학생이 팀플을 시작하기 전에 순전히 개인이 작성해둔 코드를 사용하였고 이 부분에 대해서는 조동혁 학생이 기여한 바는 없다. 이 후의 대부분의 코딩 작업에서는 함께 스터디룸에서 회의를 하고 고민하며 같이 진행하였다. 코딩 부분을 나눠서 한 것이 아니라 동시에 같은 코드 작업을 했는데 고도환 학생의 코딩 속도가 빠르고 코드의 성능이 좋았기 때문에 고도환 학생의 코드가 대부분 채택되었다. 조동혁 학생의 경우 코딩에 아예 참여를 하지 않은 것은 아니고 Minimax algorithm의 초안을 작성했고 기타 자잘한 인터페이스나 에러 들을 수정하였다.

작성된 코드에 주석을 달고, 보고서를 작성하고, 동영상을 촬영하여 everycoding사이트에 올리는 것은 조동혁 학생이 담당하였다. 물론 주석, 보고서, 동영상 모두 고도환 학생이 검토하였다. 이는 처음부터 합의된 내용이었고, 서로가 서로를 잘 보완하여 팀 프로젝트를 마쳤다고 생각하고 있다. 두 학생 모두 코드에 대해 이해하고 있고, 보고서에 어떤 내용이 들어가 있는지 알고 있다. 다만 고도환 학생의 코딩에서의 기여도가 매우 높기 때문에 팀 프로젝트를 실질적으로 이끈 것은 고도환 학생이고 상대적으로 더 큰 지분을 가지고 있다.

5. Extra Code

Search, Heuristic Function, Rule-Based 부분을 제외한 다른 모든 코드들에 대한 설명은 첨부하는 코드에 자세히 주석을 달아 두었다.

6. Source

https://en.wikipedia.org/wiki/Minimax

http://arkainoh.blogspot.com/2016/08/project.connect4.AI.html

- ➔ 코드가 아닌 아이디어만 참조함.

http://git.mrman.de/ai-c4/blob/master/s3505919.pdf

https://everycoding.net/code/ai/56