인공지능: 선배를 이겨라!

Four in a Row

Team TRIANGLE

2013210081 박진호

2013210070 김태현

2014130934 이세임

목차

1. 서론 (커넥트 4 소개)

1.1. 목표

이번 Connect 4 프로젝트의 목표는 문제를 통해 DFS 를 이용하여 우선적으로 인공지능에 대한 이해를 돕기 위함이다. 이러한 공부로 인해 수업시간에 배운 알고리즘이 실제 사례를 통해 어떠한 방식으로 사용되고 있는지 짧은 실습으로 인하여 보다 깊게 이해할 수 있게 된다. Alpha-Beta Pruning, Monte Carlo Tree Search, Rule 방법을 구현하여 완벽한 게임을 만들고자 한다.

1.2. Alpha-Beta Pruning (Minmax algorithm)

MinMax 알고리즘은 보통 두명의 플레이어가 있는 Turn 제 게임에서 사용된다. 이 알고리즘은 몇 수 앞까지 가능한 경우를 모두 가치를 매긴 다음에 상대가 최고의 수를 둔다고 가정하고 인공지능이 할 수 있는 최선의 수를 찾는 원리이다. 두명의 플레이어가 있는 게임의 목표는 처음에 시작하는 사람이 가장 큰 숫자를 가져가는 것이다. 따라서 처음에 시작하는 사람은 고를 수 있는 숫자 중 가장 큰 숫자를 고를 것이고, 다른 사람은 고를 수 있는 숫자 중 가장 작은 것(즉, 내게 불리한 수)을 고를 것이다. 게임을 진행하며 문제가 없는 한, 처음으로 시작하는 사람이 가장 큰 숫자를 얻기 위해 어떤 길을 선택해야 하는가가 게임의 문제이다. 어떻게 하면 최고의 점수를 얻을 수 있을지 알아보고자 Tree 를 탐색한다.

탐색전에 Tree 는 맨 아래 단계만 의미가 있는 수를 가진다. 탐색이 진행되면서 하위에서 상위로 값을 찾아 올라가게 되는데, 이때 상위 노드에서 선택되는 값은 현 단계가 Max 이면 아래에서 최대값, 현 단계가 Min 이면 아래에서 최소값을 선택하게 된다.

탐색 과정에서 중요한 요점은 노드 탐색에 걸리는 시간이다. 수가 늘어날수록 방문해야 할 노드가 늘어나기 때문에 메모리 사용이 상당하다. 따라서 성능이 좋은 컴퓨터일수록 빠른 계산이 가능해진다. 그렇게 노드를 일일이 방문하는 것이 많은 시간이 소요되는데, 이것을 줄이기 위한 해결책으로 Alpha-Beta Pruning 이 존재한다.

Alpha-Beta Pruning 의 pruning 은 말 그대로 '가지치기'를 의미한다. MinMax 방식으로 Tree 를 탐색하다 보면 불필요한 시간이 소요되기 때문에 이 방법을 통해 매 첫 번째 Child Node 를 검색한 후에 그 값에 따라 Pruning 을 할 수 있다. 따라서 아무리 depth 가 깊더라도 MinMax 가 계산한 시간보다 훨씬 빠른 시간 내에 같은 값을 계산할 수 있고, 오히려 더 깊이 있는 탐색을 짧은 시간에 탐색을 할 수 있다.

1.3. Monte Carlo Tree Search (MCTS)

Tree 기반 알고리즘에서 많이 사용되는 Minmax 나 Alpha-Beta Pruning 알고리즘은 게임 (오목, 체스, 커넥트포, 등) 판의 격자수가 많아지면 성능이 저하되어 오목과 같은 작은 보드게임에 많이 적용된다. 하지만 이러한 게임에서 위에 명시된 알고리즘이 사용된다 하더라도 Tree 복잡도의 깊이가 깊을수록 단순한 계산으로는 어려워지게 되기 때문에 Monte Carlo Tree Search 를 사용한다.

MCTS 방법은 수식적으로 함수를 만들어 해를 찾기가 용이하지 않을 때 적용하는데, 이는 게임에서 최선의 수를 찾기 위해 사용된다. 이 방법을 게임에 적용하기 위해서는 3 가지 조건을 만족시켜야 한다.

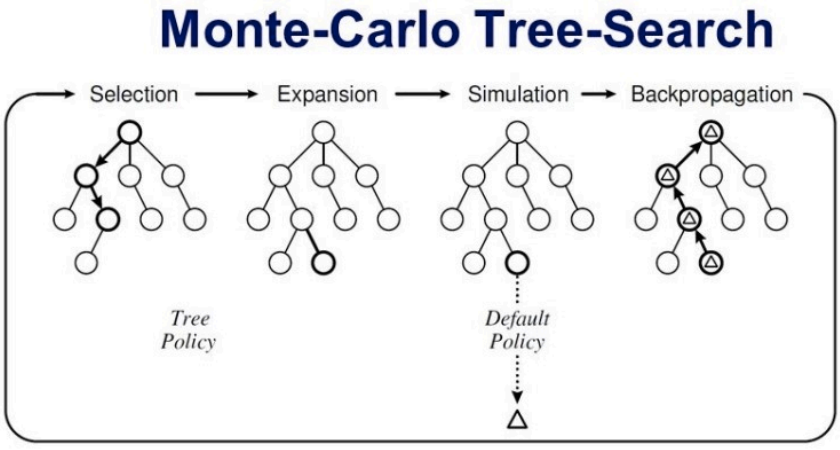
(1) 게임의 최대/최소 점수 값이 있어야하고

(2) 게임 규칙이 정해져 있으며 Perfect information 게임이어야 하며

(3) 게임의 길이가 제한되어 시뮬레이션이 비교적 빠른 시간내에 끝나야 한다. [1]

이 3 가지 조건을 만족시킨다는 가정하여 MCTS 는 최선의 선택을 내리기 위한 방법으로 Tree 에서의 Random Simulation 을 통하여 결정하는데 이러한 과정은 총 4 단계로 나눈다:

[1] https://senseis.xmp.net/



(1) 선택 (Selection) 에서는 Root Node 에서 시작하여 현재까지 펼쳐진 Tree 를 선택한다.

(2) 확장 (Expansion) 에서는 만약 전 단계에서 선택한 Tree 에서 게임의 종료가 되지 않은 경우에는 한 개 이상의 Child Node 를 생성하여 하나를 선택한다.

(3) 시뮬레이션 (Simulation) 에서는 확장 단계에서 선택된 Child Node 에서 게임의 시뮬레이션을 돌려 게임이 종료 될 때까지 시행한다.

(4) 역전파 (Backpropagation) 에서 선택된 Child Node 에 시뮬레이션 결과를 반영한다. [2]

MCTS 에서 보다 효율적인 검색이 가능하게 하는 방법중에 UCT 는 기본기능에 해당된다고 볼 수 있다. 특히 첫번째 단계인 '선택' 과정에서 UCT 를 이용하며 기준을 둔다. UCT (Upper Confidence bounds applied to Trees)란 exploitation 과 exploration 의 적절한 조화를 통하여 계산된 식을 이용하여, 가장 큰 값을 선택하게 하여 이기는 경우가 많은 노드를 확실하게 더 많이 선택할 수 있게 필요로하는 기준 값이라 볼 수 있다.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

여기서 explotation 에 해당되는 Wi 는 해당 node 에서 이긴 횟수를 의미하고 ni 는 해당 node 에서 수행한 게임 횟수를 의미한다. 또한 exploration 에 해당되는 C 는 상수이고 ti 는 총 simulation 을 한 횟수를 나타낸다. 만약 UCT 방법을 이용했을 때, 선택한 node 가 한 번도 수행하지 않았던 node 라면 그대로 시뮬레이션을 진행하고, 만약 한번 실행된 node 라면 확장시킨 뒤 하위 child node 들을 다시 UCT 값과 비교하여 선택하고 시뮬레이션 해야한다. [3]

2. 본론 (알고리즘 소개)

2.1. 게임 설명

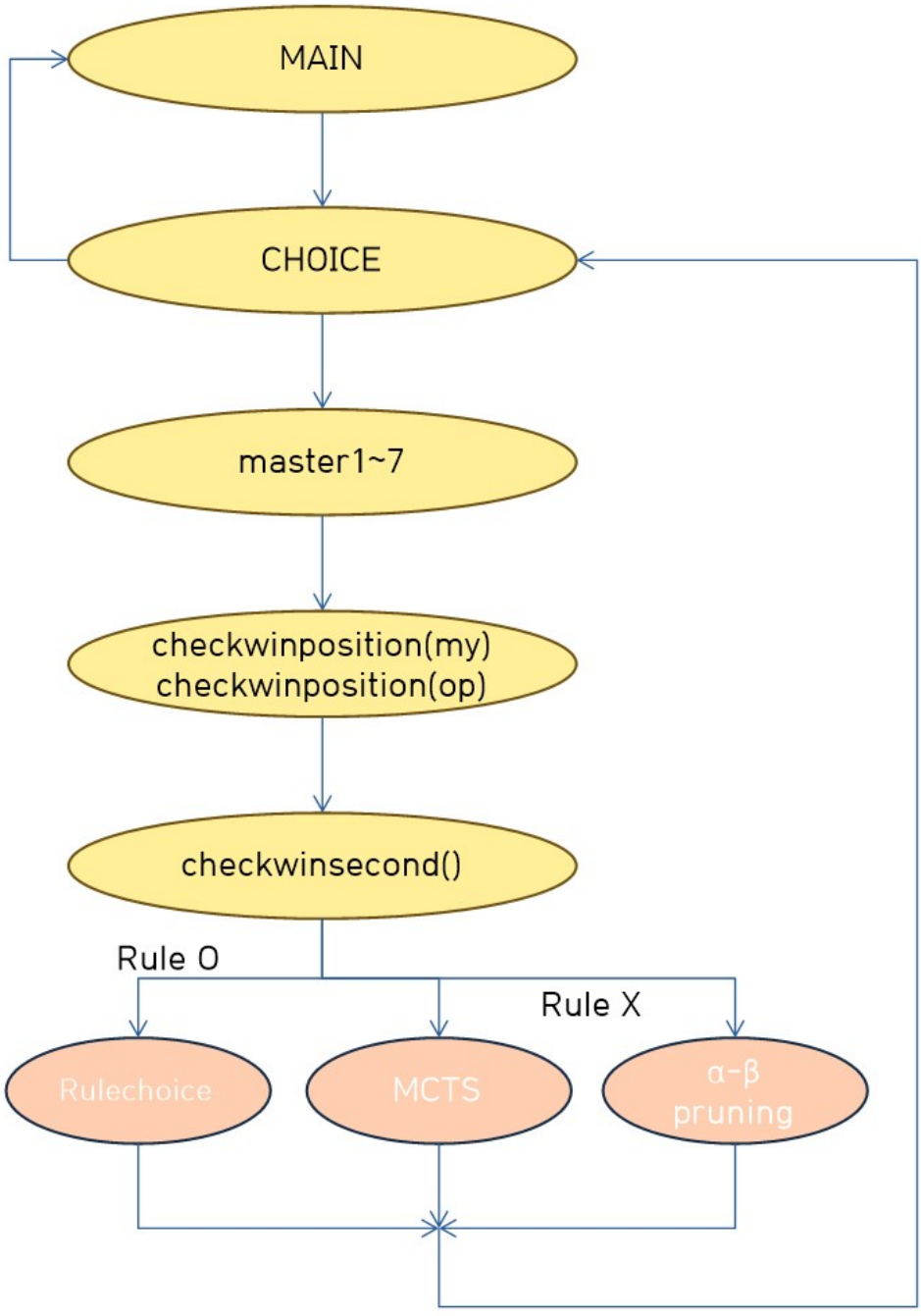
Connect 4 게임은 쉽게 말하면 4 목이라고 할 수 있다. 최근 알파고와의 대결로 인해 많은 주목을 받았던 오목과 다른 점은 평면에 플레이어가 놓고 싶은 위치에 놓을 수 없고, 입체로 게임판을 세워서 넣는 것이다. 이 게임은 가로 7 칸, 세로 6 칸인 직사각형 판을 세워서 말을 떨어뜨려 2 명의 플레이어가 교대로 말을 두면서 가로, 세로, 대각선 중 어떠한 방향으로든 먼저 본인 말 4 개가 일직선으로 이어지게 만들면 승리한다. 이 게임은 Victor Allis 와 James Allen 에 의해 첫 수가 가운데 줄에 놓아지면 41 번째 수가 되기 전에 반드시 승리한다는 것이 증명되었다. 그 옆 두 칸에 두면 첫 수를 둔 선수가 비기도록 만들 수 있고, 첫 수가 나머지 4 줄인 경우 지는 수가 존재한다.

---

[2] http://mcts.ai/about/index.html

[3] 수업자료

2.2. 코드

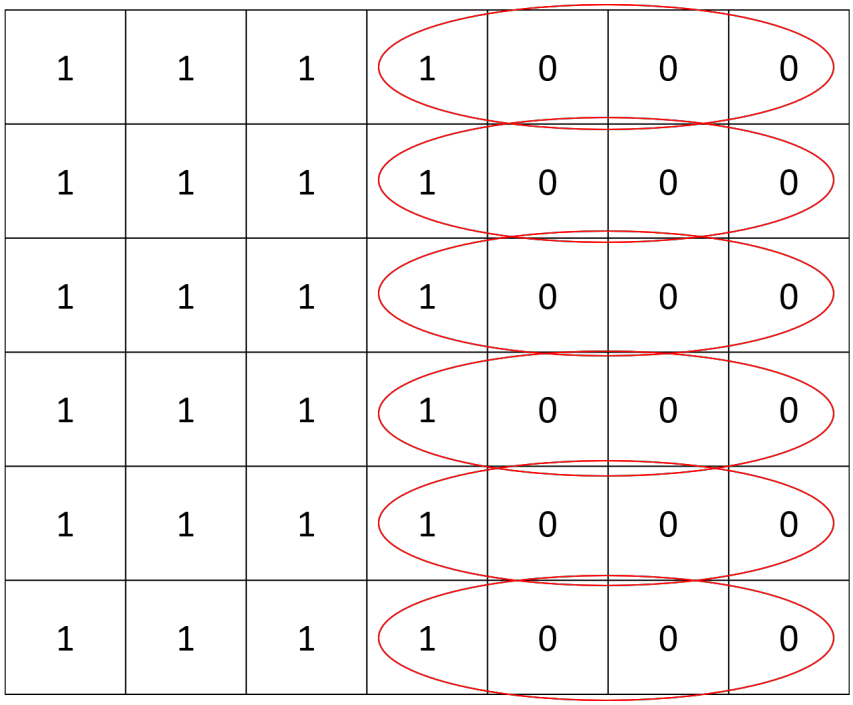


Bit

__int64 current_state, my_state, opp_state; // 전체 보드상황 (있으면 1 없으면 0), 내(컴퓨터) 상황, 상대(유저) 상황
__int64 mask1 = 0b1111000111100011110001111000111100011111000; // 가로 승리확인 제한 mask
__int64 mask2 = 0b1111111111111111111111000000000000000000000; // 세로 승리확인 제한 mask
__int64 mask3 = 0b1111000111100011110000000000000000000000000; // 오른쪽 위 승리확인 제한 mask
__int64 mask4 = 0b0001111000111100011110000000000000000000000; // 왼쪽 위 승리확인 제한 mask
int bottom[7] = { 41, 40, 39, 38, 37, 36, 35 }; // 보드에 바로 놓을 수 있는 각 위치 별 높이에 대한 left shift value.

```
int record[44]; // 기보 기록.
int eval_board[42] = { 3, 4, 5, 7, 5, 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, 5, 4, 3 }; // evaluation에 쓰이는 각 칸 별 점수.
```

Board 를 표현하는 가장 효율적인 방법으로 비트를 선택했다. 기본적으로 2 차원 배열은 7x6 에 대한 메모리가 필요한데, 비트로 사용하게 될 경우 _int64 형 변수 하나만으로 board 의 state 를 표현할 수 있기 때문이다. 또한, 승리 조건을 판별하는 check 함수에서 3 중 for 문을 통해 탐색하지 않고, 7 번만의 bit 연산으로 승리 여부를 판별할 수 있기 때문에 훨씬 빠른 속도로 처리가 가능하다. 이곳에서 mask1 은 가로로 연속된 숫자로 표현되기 때문에 이 board 에 경계가 존재하지 않는다. 따라서 한 줄의 끝과 그 다음 줄 맨 앞이 같은 줄에 있다고 인식할 가능성이 있어, mask 를 통해 각 방향으로 4 개가 이어질 수 있는 영역을 구분하였다.



위 예시를 통해 mask1 이 어떠한 방식으로 적용되는지 볼 수 있다.

Main

```
int main() {
        int mode, ch, count = 1;
        _int64 t = 1;
        mode = 0;
        printf("먼저 하시겠습니까? 예(1) 아니오(2)\n");
        scanf("%d", &mode);
        while (mode != 1 && mode != 2) {
                printf("\n 예(1) 아니오(2) 중 하나를 입력하세요\n");
                scanf("%d", &mode);
        }
        prtboard(my_state, opp_state);
        if (mode == 1) {
```

```
                ch = getchoice();
                record[count] = ch;
                ch -= 1;
                current_state |= t << bottom[ch];
                opp_state |= t << bottom[ch];
                bottom[ch] -= 7;
                prtboard(my_state, opp_state);
                count++;
        }
        while (1) {
                printf("룰을 적용할까요? 예(1) 아니오(2)\n");
                scanf("%d", &mode);
                ch = choice(count, mode);
                current_state |= t << bottom[ch];
                my_state |= t << bottom[ch];
                bottom[ch] -= 7;
                prtboard(my_state, opp_state);
                record[count] = ch + 1;
                count++;
                if (count == 43) {
                        printf("비겼습니다!");
                        break;
                }
                if (checkwin()) break;
                ch = getchoice();
                ch -= 1;
                current_state |= t << bottom[ch];
                opp_state |= t << bottom[ch];
                bottom[ch] -= 7;
                prtboard(my_state, opp_state);
                record[count] = ch + 1;
                count++;
                if (count == 43) {
                        printf("비겼습니다!");
                        break;
                }
                if (checklose()) break;
        }
        printf("\n 종료하려면 아무 키나 입력해 주세요.");
        scanf("%d", &mode);
        return 0;
}
```

전체적인 게임을 진행하기 위한 순서가 존재한다. 이 부분에서는 게임이용자에게 Turn 에 대한 순서 여부를 물은 다음, 게임에 Rule을 적용할 것인지 적용하지 않을 것인지를 선택하게 된다. 순서를 정하고 게임을 시작하는 순간부터 prtboard 를 통해 게임 보드 현황을 출력한다. 그 다음 상대방이 말을 둘 때까지 입력 값을 기다린다. 입력을 하다보면 승부의 결정이 나기 때문에 check( )로 한쪽이 이겼을 때 게임을 종료시키거나, count=43 즉, 43 번째의 말을 둘 차례가 오게 되면 draw, 게임을 비기게 된다.

Check

```
_int64 check(_int64 state) { // 현재 입력받은 state에 가로 세로 대각선으로 연속한 4개의 돌이 놓여있는 부분이 있는지를 확인하는 함수.
	_int64 check;
	check = state & state << 1 & state << 2 & state << 3 & mask1; // 가로
	check |= state & state << 7 & state << 14 & state >> 7 & mask2; // 세로
	check |= state & state << 8 & state << 16 & state >> 8 & mask3; // 오른쪽 위 대각선
	check |= state & state << 6 & state << 12 & state << 18 & mask4; // 왼쪽 위 대각선
	return check;
}
```

Check 함수에서 bit 연산을 하는데 예를 들어서 가로를 확인할 때 temp = state & 0<<1 & 0<<2 & 0<<3 를 통해 알아 볼 수 있듯이 bit 연산을 left shift 최대 3 번을 통해 4 개를 and 시켜 1 이 나오는 경우가 하나이기 때문에, 이것이 존재한다면 1 이 있는 지점에서부터 오른쪽 3 칸이 전부다 1 이라는 것을 의미한다. 따라서 이 결과로 우리는 1 이 있는 비트는 전부 다 1이었기에 결국 게임에서 이긴 지점이 있다는 것으로 temp 를 통해 승리 여부를 판단할 수 있다. 반면에, 하나라도 네개의 1 이 연속된 곳이 없다면, left shift 를 통해 절대 값이 1 이 나올 수 없어, temp 의 결과 갑이 0 이 되어 승리를 거두지 않았다는 것이다.

Choice

```
int choice(int count, int mode) { // main 에서 choice 를 요청하면 현재 내(컴퓨터)가 어디에 두어야 할 지를 리턴한다. 룰로 두는 경우도 포함.
	int i, bet[7], temp;
	for (i = 0; i < 7; i++) bet[i] = bottom[i];
	if (count == 1) { // count 1~7 은 정해진 퍼펙트 솔루션대로 수를 둔다.
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return 2;
	}
	else if (count == 2) {
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return master2[record[count - 1]] - 1;
	}
	else if (count == 3) {
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return master3[record[count - 1]] - 1;
	}
	else if (count == 4) {
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return master4[record[count - 3]][record[count - 1]] - 1;
	}
	else if (count == 5) {
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return master5[record[count - 3]][record[count - 1]] - 1;
	}
	else if (count == 6) {
		if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
		return master6[record[count - 5]][record[count - 3]][record[count - 1]] - 1;
	}
	else if (count == 7) {
```

```
                if (mode == 1) printf("RULE : 정해진 master solution 에 의해 둔다.");
                return master7[record[count - 5]][record[count - 3]][record[count - 1]] - 1;
        }
        else {
                temp = checkwinposition(my_state, bet); // 퍼펙트 솔루션을 제외하고 먼저 한 수만에 이길 수 있는 수가 있다면 둔다.
                if (temp < 7) {
                        if (mode == 1) printf("RULE : 이거 두면 내가 이기지?ㅎㅎ (내가 한 번에 이기는 수를 둔다.)");
                        return temp;
                }
                temp = checkwinposition(opp_state, bet); // 상대방이 한 수만에 이길 수 있는 수가 있다면 막는다.
                if (temp < 7) {
                        if (mode == 1) printf("RULE : ㅎㅎ내가 모를 줄 알았어? (상대방이 한 번에 이기는 수를 막는다.)");
                        return temp;
                }
                temp = checksecondwin(); // 내가 두 수만에 이길 수 있는 수가 있는지, 상대방이 두 수만에 이길 수 있는 수가 있는지 판별하고 공격하거나 방어한다.
                if (temp != -1) {
                        if (mode == 1) printf("RULE : 내가 두 번만에 확실히 이기는 수를 둔다. / 상대방이 두 번만에 확실히 이기는 수를 막는다.");
                        return temp; // temp 가 -1 이면 그런 수가 없다는 것을 의미한다.
                }
                if (mode == 1) {
                        printf("RULE : evaluate_board 를 기준으로 높은 점수가 있는 곳 ( 중앙에 가까운 곳 ) 에 둔다.");
                        return rulechoice(count, bet); // 룰로 두어야 하는 경우 호출하여 결과를 리턴한다.
                }
                else {
                        struct ret x;
                        x = search(current_state, my_state, opp_state, bottom, 1, 1, -2147483647, 2147483647); // 룰을 적용시키지 않는 경우, 알파베타 프루닝을 진행한다.
                        //x = MCTS(current_state, my_state, opp_state, bottom, count);
                        if (x.choice == -1) {
                                for (i = 0; i < 7; i++) {
                                        if (bottom[i] >= 0) return i;
                                }
                        }
                        return x.choice;
                }
        }
}
```

게임보드의 현황을 파악한 후 어디에 말을 놓아야 승리의 길을 걸을 수 있을지 보는 과정이다. Choice 로 들어오면서 main 의 count (턴 수)값과 mode (룰 적용 여부)의 함수를 받은 후, 만약 mode 가 rule 함수를 호출해서 입력값을 받으면, master 배열에 입력된 값으로 수를 둔다. 하지만, rule 을 사용하지 않으면 master 배열에 입력된 값을 return 하고, Alpha-Beta Pruning 이나 MCTS 를 진행한다.

Master

```
int master2[8] = { 0, 4, 3, 4, 4, 4, 5, 4 }; // master'n'은 n번째 수에 대해 두어야 할 perfect한 수를 pre-based 해 놓은 배열.
```

```
int master3[8] = { 0, 4, 6, 2, 3, 3, 2, 3 };
int master4[8][8] =
{ { 0,0,0,0,0,0,0,0 },{ 0,4,4,4,4,4,4,3 },{ 0,3,2,3,2,3,3,3 },{ 0,4,4,3,4,3,4,4 },{ 0,3,3,2,4,3,5,4 },{ 0,4,4,4,4,5,4,4 },{ 0,5,5,5,4,5,6,5 },{ 0,3,4,4,4,4,4,4 } };
int master5[8][8] =
{ { 0,0,0,0,0,0,0,0 },{ 0,5,2,5,5,5,2,5 },{ 0,3,6,3,6,3,6,2 },{ 0,3,4,4,4,3,4,4 },{ 0,3,3,4,3,3,3,3 },{ 0,3,3,5,3,3,3,3 },{ 0,2,4,4,4,4,4,4 },{ 0,3,3,4,3,3,3,3 } };
int master6[8][8][8] = {{{…}…}…}
int master7[8][8][8] = {{{…}…}…}
```

Master 'n'은 n 번째 수에 두어야할 perfect 한 수를 pre-based 해 놓은 배열이다. Master 1 부터 7 까지 존재하는데 그 이유는 7 수까지 보게 된다면 우리 프로그램이 게임을 먼저 시작했을 때 상대방이 3 번의 수를 둘 수 있기에 7^3=329 개이다. 이 이상 사람이 손으로 구현해내기에 소요시간이 너무 크고, 오차 가능성이 존재하기 때문이다. Master 를 사용하는 이유는 게임을 진행하는데에 있어서 Full Tree 를 하지 못하는 상황에서 Partial Tree 를 진행하게 되면 초반에 evaluating 결과값이 알고리즘 상에는 문제가 없지만 최적 값을 보이지 않는 경우가 빈발했다. 따라서 결과적으로 초반에는 경우의 수가 상대적으로 적기 때문에 우리 프로그램이 최적의 솔루션을 미리 만들어 두게 되면 전체적인 방향성이 우리한테 유리하게 돌아가게 되므로 오차범위가 줄어들 것이다. 그의 이유는 게임 초반에 두는 수일수록 Full Tree 를 그릴 때 search depth 가 크기에 불확실성이 높기 때문이다.

Checkwinposition(my)

```
int checkwinposition(__int64 my_state, int bot[7]) { // 1~7번에 한 번씩 수를 둔 후 check함수를 통해 승패가 갈리는 지 여부를 판단하여 있으면 0~6(놓을 위치), 없으면 7 리턴.
	__int64 t=1;
	int i;
	for (i = 0; i < 7; i++) {
		if (bot[i] >= 0) {
			if (check(my_state | t << bot[i])) break;
		}
	}
	return i;
}
```

입력받은 state 에 대해서 그 state 에 함수를 둔 다음, 게임이 끝날 수 있는지를 찾아서 그 위치를 돌려주는 함수이다. 여기서 게임이 끝난다는 것은, my (즉, 우리 프로그램)가 게임을 승리한다는 것이다.Checkwinposition 의 프로세스 의미는 우리 프로그램이 이길 수 있는 곳을 찾고, 만약 이길 수 있는 자리가 있다면 그 곳에 말은 둔다. 그렇다면 결과값을 choice 로 return 하고, 의미있는 결과값(0 에서 6)이라면, 그 값을 받아 main 으로 return 한다. 만약 이 함수를 이용하고도 프로그램이 이길 수 있는 자리를 찾기 못한다면, checkwinposition(opp)로 넘어가게 된다.

Checkwinposition(opp)

전 단계에서, checkwinposition(my) 함수를 이용했음에도 불구하고 우리 프로그램이 본인의 turn 이었을 때 의미 있는 값(7)을 찾지 못했을 경우 checkwinposition(opp)를 호출하게 된다. 따라서 프로그램이 바로 이길 수 없다면 이 함수를 사용하여 상대방이 바로 이길 수 있는지 확인한 다음, 만약 존재한다면 방어를 하게 된다.

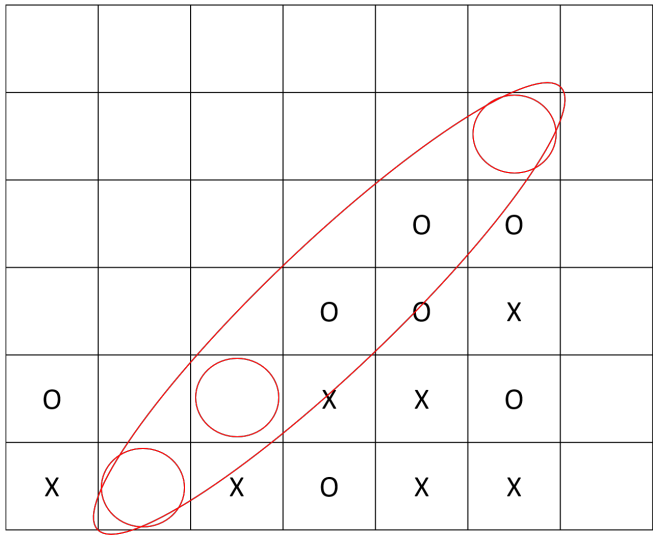
Checksecondwin

```
int checksecondwin() { // 00110, 01010, 01100 중 한 경우의 수가 존재하는지 판단한다. 이는 두 번의 수 만에 확실히 이길 수 있는 경우에 속한다.
	int i, j, my, opp, choice = -1;
```

```
	int b[10][10];
	_int64 temp1 = my_state, temp2 = opp_state;
	for (i = 0; i < 6; i++) { // 쉽게 판단하기 위해 임시로 보드로 변환.
		for (j = 6; j >= 0; j--) {
			my = temp1 % 2;
			temp1 /= 2;
			opp = temp2 % 2;
			temp2 /= 2;
			if (my == 1) b[i][j] = 1;
			if (opp == 1) b[i][j] = 2;
			if (my == 0 && opp == 0) b[i][j] = 0;
		}
	}
	for (j = 0; j < 7; j++) b[7][j] = -1; // 바닥에서 index error를 방지하기 위해 한 줄을 더 두고 -1로 바닥임을 표시한다. !=0인 조건에 걸리지 않게 하기 위함도 있다.
	for (i = 0; i < 6; i++) {
		for (j = 0; j < 7; j++) { // 가로로 5개의 수를 봐야 하므로 중앙을 기준으로 좌 우를 본다고 할 때 j가 2~4 ( 3~5 ) 일 경우만 확인해야 한다.
			if (j >= 2 && j <= 4 && b[i][j] == 0 && b[i + 1][j] != 0 && b[i][j - 1] == 1 && b[i][j + 1] == 1 && b[i][j - 2] == 0 && b[i + 1][j - 2] != 0 && b[i][j + 2] == 0 && b[i + 1][j + 2] != 0) return j;
			if (j >= 2 && j <= 4 && b[i][j + 1] == 0 && b[i + 1][j + 1] != 0 && b[i][j - 1] == 1 && b[i][j] == 1 && b[i][j - 2] == 0 && b[i + 1][j - 2] != 0 && b[i][j + 2] == 0 && b[i + 1][j + 2] != 0) return j + 1;
			if (j >= 2 && j <= 4 && b[i][j - 1] == 0 && b[i + 1][j - 1] != 0 && b[i][j] == 1 && b[i][j + 1] == 1 && b[i][j - 2] == 0 && b[i + 1][j - 2] != 0 && b[i][j + 2] == 0 && b[i + 1][j + 2] != 0) return j - 1;
			if (i >= 2 && i <= 3 && j >= 2 && j <= 4) { // 대각선을 보는 경우는 i가 2~3, j가 2~4안에 있어야 보드를 벗어나지 않고 확인할 수 있다.
				if (b[i + 2][j - 2] == 0 && b[i + 3][j - 2] != 0 && b[i - 2][j + 2] == 0 && b[i - 1][j + 2] != 0) {
					if (b[i][j] == 0 && b[i + 1][j] != 0 && b[i + 1][j - 1] == 1 && b[i - 1][j + 1] == 1) return j;
					if (b[i][j] == 1 && b[i + 1][j - 1] == 1 && b[i - 1][j + 1] == 0 && b[i][j + 1] != 0) return j + 1;
					if (b[i][j] == 1 && b[i + 1][j - 1] == 0 && b[i + 2][j - 1] != 0 && b[i - 1][j + 1] == 1) return j - 1;
				}
				if (b[i + 2][j + 2] == 0 && b[i + 3][j + 2] != 0 && b[i - 2][j - 2] == 0 && b[i - 1][j - 2] != 0) {
					if (b[i][j] == 0 && b[i + 1][j] != 0 && b[i - 1][j - 1] == 1 && b[i + 1][j + 1] == 1) return j;
					if (b[i][j] == 1 && b[i - 1][j - 1] == 1 && b[i + 1][j + 1] == 0 && b[i + 2][j + 1] != 0) return j + 1;
					if (b[i][j] == 1 && b[i - 1][j - 1] == 0 && b[i][j - 1] != 0 && b[i + 1][j + 1] == 1) return j - 1;
				}
			} // ==0 임을 확인하는 부분에서 꼭 !=0 인지 여부를 확인해야 한다. 바로 둘 수 있는 경우에만 확실한 승리수라고 생각할 수 있기 때문이다.
		}
	}
	for (i = 0; i < 6; i++) { // 내 확실한 수를 확인한 후 없으면 상대방의 확실한 수를 확인한다. 과정은 동일하다.
```

```
                for (j = 0; j < 7; j++) {
                        - 위와 동일하게 수행.
                }
        }
        return choice;
}
```

마찬가지로, 전 단계에서 상대방이 이길 수 있는 경우를 찾기 못했을 때 checksecondwin 으로 넘어오게 되는데 보드에 I,j,1,2 를 사용하게 된다. 이 단계에서는 다음 3 가지와 같은 방식으로 가로 또는 대각선에 존재하는지 파악하게 된다: _ O _ O _ , _ O O _ _, _ _ O O _.



위 예시를 보았을 때, _ _ O O _ 와 같은 경우를 나타내는데, 우리는 _ O O O _ 으로 수를 두면 무조건 승리한다. 하지만 두번째 빈칸에 말을 두기 전에 그 아래칸에 말 (나 또는 상대방, 무관)이 있어야 원하는 자리에 말을 둘 수 있다. 이 조건을 확인하기 위해 중간에 있는 말을 i,j 로 기준을 둔 다음 그 자리에서 +,-1 또는 2 를 하며 과정을 거치게 하였다. 따라서 총 5 칸의 상태가 바로 둘 수 있을 만큼 아래칸이 꼭 차 있어야 한다.

MCTS

```
ret MCTS(__int64 state, __int64 m_state, __int64 o_state, int bot[7], int count) { //MCTS를 진행.
        int i, j, bet[7], val, ct=0, max=-9999;
        __int64 t = 1, temp;
        ret re;
        for (i = 0; i < 7; i++)
                bet[i] = bot[i];
        for (i = 0; i < 7; i++) { // 7개의 수를 둔 경우를 가정.
                temp = t << bet[i];
                bet[i] -= 7;
                ct = 0;
                for(j=0; j<1000000; j++){ // 100만번의 수를 둔다
                        val = randomgo(state | temp, m_state | temp, o_state, bet, -1, count+1);
                        if (val >= 0) ct++; // 100만번 중 이기거나 비기는 경우를 카운트 한다. 이상적인 인공지능의 경우
비기는 것이 최대 결과값이므로 포함시켰다.
                }
                if (ct > max) {
```

```
                max = ct;
                re.choice = i;
            }
            bet[i] += 7;
        }
        re.value = 0;
        return re;
}
```

게임을 하면서 룰을 적용하지 않을 경우 checkwin 을 수행한 다음 MCTS 알고리즘으로 넘어오게 된다. 여기서 첫번째 칸에 말을 두는 순간, for 문을 돌면서 randomgo 를 호출한다. 여기서 Temp 를 받고, 만약 이겼다면 999, 졌으면 -999 그리고 비겼다면 0 을 표시한다. 한 번 randomgo 를 돌렸을때 승부가 들어있다 볼 수 있다. 즉, 이기거나 비기면 count 를 증가시키는데, 이것은 randomgo 를 시행할 때 백만번을 호출하여 수행중에서 이기는 수를 ct 에 적게되는 것이다. 그렇게 각 열마다 백만번씩 돌려서 퍼센트가 제일 높은 곳이 승률이 높다는 것을 알 수 있다.

Alpha-Beta pruning

```
struct ret search(__int64 state, __int64 m_state, __int64 o_state, int bot[7], int depth, int minmax_state, int alpha, int beta) {
        int i;
        __int64 t=1, temp;
        struct ret ge, re;
        if (minmax_state == 1) { // 내 차례인 경우 상대방이 이겼다면 더 이상 탐색을 진행하지 않는다.
            temp = check(o_state);
            if (temp) {
                re.value = -99999999;
                re.choice = -1;
                return re;
            }
        }
        else { // 반대로 상대방 차례인 경우 내가 이겼다면 더 이상 탐색을 진행하지 않는다.
            temp = check(m_state);
            if (temp) {
                re.value = 99999999;
                re.choice = -1;
                return re;
            }
        }
        if (depth == 12) { // depth제한을 두어 시간제한을 둔다.
            re.value = evaluation_state_board(m_state, o_state); // 최대 depth에 도달했을때 알파베타 프루닝에 사용할 값을 현재 state를 기준으로 evaluation해 부여한다.
            re.choice = -1;
            return re;
        }
        re.value = -2100000000;
        re.value *= minmax_state;
        re.choice = -1;
        for (i = 0; i < 7; i++) {
            if (bot[i] >= 0) {
                temp = t << bot[i];
```

```
                        bot[i] -= 7;
                        if (minmax_state == 1) { // 내 차례를 의미한다. MINMAX의 MAX에 해당된다.
                                ge = search((state|temp), (m_state|temp), o_state, bot, depth + 1, minmax_state*(-1),
alpha, beta);
                                bot[i] += 7;
                                if (ge.value == 99999999) {
                                        re.value = 99999999;
                                        re.choice = i;
                                        return re;
                                }
                                if (ge.value > re.value) {
                                        re.value = ge.value;
                                        re.choice = i;
                                }
                                if (ge.value > alpha) alpha = ge.value;
                                if (alpha >= beta) break;
                        }
                        else { // 상대방 차례를 의미한다. MIN에 해당된다.
                                ge = search((state|temp), m_state, (o_state|temp), bot, depth + 1, minmax_state*(-1),
alpha, beta);
                                bot[i] += 7;
                                if (ge.value == -99999999) {
                                        re.value = -99999999;
                                        re.choice = i;
                                        return re;
                                }
                                if (ge.value < re.value) {
                                        re.value = ge.value;
                                        re.choice = i;
                                }
                                if (ge.value < beta) beta = ge.value;
                                if (alpha >= beta) break;
                        }
                }
        }
        return re;
}
```

Alpha-Beta pruning 을 이용하게 되는 경우는 MCTS 와 마찬가지로 게임에 룰을 적용하지 않았을 때이다. 이 알고리즘을 통해 우리는 방금 상대방이 둔 수가 게임을 끝냈는지 아닌지 확인할 수 있고, 만약 이겼다면 99999999 이고, 졌을 경우엔 -99999999 이다. 여기서 depth 를 12 로 정해놓은 이유는, 만약 12 로 정해놓지 않는다면 게임이 끝날 때까지 depth 탐색이 진행된다. 어느 한 쪽의 승리로 게임이 끝나거나, 더 이상 말을 놓을 자리가 없어서 끝나거나, 경우의 수가 많아서 12 개의 말만 두었을 때에 판단하기 위해서 조건에 따라 가지 쳐서 돌지 않는 것이다. 12 번째 말에 도달하게 되면 밑을 탐색해서 결과값을 받아야하는데, 그 값을 직접 계산하기엔 소요시간이 오래 걸리기 때문에 evaluation function(이 수가 나한테 유리 또는 불리한지)을 사용하며 예측을 하게 된다. 이러한 결과치로 state 에 대한 점수를 주는 것이다.

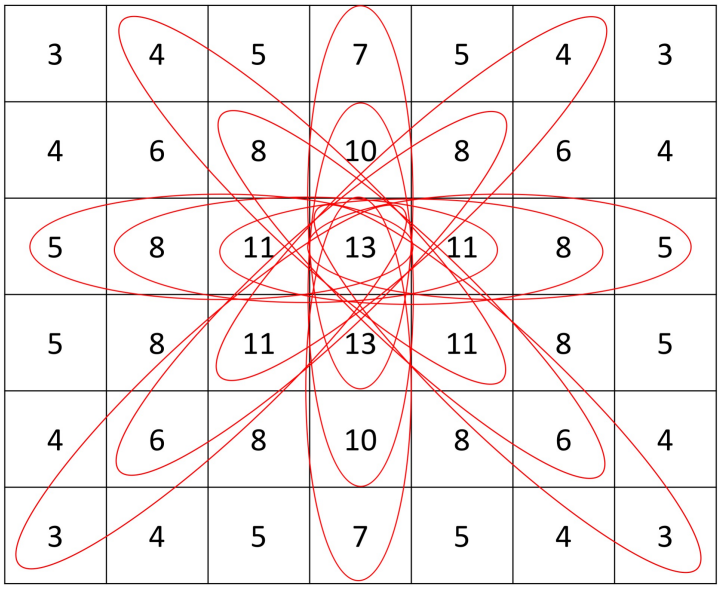
뿐만 아니라 말을 두기 위해 최고 값을 사용해야 함으로 우리 프로그램은 max 값을 사용해야하기에 최저 값과 비교하여 찾고, 상대방 차례일 때에는 min 값을 사용해야하기 때문에 최고 값과 비교해야한다.

따라서 여기서 ge.value 를 통해 가지 쳐서 시간 단축을 하며 상대방 입장에서 내가 이기는 수이면 말을 둘 일이 없으니 말을 두지 않고, ge.value < re.value 로 Minmax 를 적용하며 이기기 위한 max 값을 선택한다.

evaluation_state_board

```
int eval_board[42] = { 3, 4, 5, 7, 5, 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, 5, 4, 3 }; // evaluation 에 쓰이는 각 칸 별 점수.
int evaluation_state_board(__int64 m_state, __int64 o_state) { // eval_board를 기반으로 현재 state의 전세를 score로 변환하여 리턴.
        int i = 0, m_score = 0, o_score = 0;
        for (i = 0; i < 42; i++) {
                m_score += eval_board[i] * (m_state % 2);
                m_state /= 2;
                o_score += eval_board[i] * (o_state % 2);
                o_state /= 2;
        }
        return m_score - o_score;
}
```

게임의 state 를 평가하기 위해 다음과 같은 board 를 작성해보았다. Connect 4 의 게임 룰인, 총 4 개의 말이 가로, 세로 또는 대각선에 연속으로 존재하면 이기는 것이다. 따라서 평가 점수는 많은 승리를 거둘 수 있는 자리일수록 숫자가 높아진다. 아래의 예시를 보면 숫자 13 을 둘러싼 빨간 oval 이 보인다.



이것이 4 개의 말이 함께 존재할 때 그려진 것이다. 즉, 13 을 기준으로 13 개의 4 목이 있기 때문에 그 칸의 점수가 13 인 것이고, 만약 한 칸에 숫자 5 가 있다면 그 자리에서는 4 목이 5 개가 존재할 수 있기 때문에 점수가 5 인것이라 볼 수 있다.

2.3. Rule

게임을 시작할 때나 하는 도중, 룰을 적용시켰을 때:

(1) 앞의 몇 수를 master solution 으로 둔다 (pre-based).

(2) 우리가 한 turn 만에 이기는 수를 둔다.
(3) 상대가 한 turn 만에 이기는 수를 막는다.
(4) 우리가 두 turn 만에 이기는 수를 둔다.
(5) 상대가 두 turn 만에 이기는 수를 막는다.
(6) Evaluation board 값을 기준으로 높은 점수에 둔다.

3. 결론

3.1. 코드 성능 분석

처음 프로그램이 실행되면 main 에서 전반적인 게임을 진행한다. getchoice 함수와 prtboard 함수를 통해 게임의 상태에 대한 입출력을 진행했으며, 턴을 카운트하고 기보를 기록하며 룰의 적용여부 등을 입력받는다. 그리고 우리 프로그램이 말을 두어야 하는 경우에 choice 함수를 호출하여 어느 곳에 말을 두는 것이 가장 이상적인지를 계산하고 말을 둘 위치값을 받는다 (1~7). choice 함수는 아래와 같은 순서로 값을 리턴한다.

(1) 최우선적으로 첫번째~일곱번째 수에 대해 미리 계산된(pre-based value) master 배열의 값을 리턴한다.
(2) 우리가 바로 (하나의 말로) 이길 수 있는 수가 있다면 해당 위치에 말을 둔다.
(3) 상대방이 하나의 말로 이기는 수가 있다면 해당 위치에 말을 두어 방어한다.
(4) 가로, 대각선 방향에 01010 / 01100 / 00110 (1 은 나, 0 은 빈칸) 의 형태가 존재하는지 확인한다. 이 때, 빈칸은 바로 아래칸에 다른 말이 놓여있는 경우에만 해당된다. 이 조건을 만족할 때, 저 수는 두 수만에 무조건 승리할 수 있는 수가 된다. 따라서 조건을 만족하면 가운데 세 칸 중 하나 있는 빈칸에 수를 둔다.
(5) 4 번과 같은 형태로 상대방의 말이 놓여있는 지 확인한다. 있을 경우, 가운데 세 칸 중 비어 있는 한 칸에 수를 두어 막는다.
(6) (1) 룰을 적용시켜야 하는 경우, evaluate_board 를 기준으로 가장 높은 점수에 해당하는 위치에 말을 둔다.
(2) 룰을 적용시키지 않는 경우, search 함수를 통해 DFS 를 수행한다. search 함수는 지정된 depth 를 기준으로 말단 노드에서 evaluation function 을 통해 해당 노드의 값을 얻고 그 값들을 기준으로 alpha-beta prunning 을 적용시킨 minmax algorithm 을 수행한다. 이를 통해 얻게된 최적의 값으로 현재 state 에서의 선택지를 고르고 해당 값을 리턴한다.
(3)룰을 적용시키지 않는 경우 + MCTS 를 사용하는 경우는 1~7 의 위치를 하나씩 고르면서 각 위치에 말을 두는 것을 가정하며 진행한다. 한 수를 둔 이후에는 randomgo 함수를 통해 random 하게 말을 두어 게임이 끝나는 경우까지 진행하는데, 이 과정을 100 만번 반복한다. (수행시간에 따라 그 횟수는 달라질 수 있다.)

3.2. 시행착오 및 문제점 개선 :

(1) 가장 처음 프로그램을 작성함에 있어서 목표로 두었던 것은 퍼펙트 솔루션을 구현하는 것이었다. 퍼펙트 솔루션을 구현하기 위해 시도하고자 했던 방법은 Full-Tree 를 구현하는 것과 완벽한 evaluation function 을 구현하는 것이었다.
(2) Full-Tree 를 구현하는 방법으로는 직접 프로그램을 돌려 결과를 기록하고 그 결과를 기반으로 데이터베이스를 구축해 모든 경우의 수에 대한 perfect solution 을 찾아내거나 인터넷에서 미리 구해져있는 자료를 받아 쓰는 것이었다. 하지만 직접 프로그램을 돌리는 것은 프로그램 구현시간을 제하고 제출기한까지의 시간이 얼마 남지 않아 불가능했고 인터넷에서 별도의 자료를 찾는 것도 어려웠다. 애초에 별도의 데이터베이스를 구현하는 것도 쉽지 않아서 포기할 수 밖에 없었다.
(3) 차선책으로 생각한 완벽한 Evaluation function 을 찾는 시도는 인터넷에서 다양한 방식의 evaluation function 을 참고하여 사용해 보았으나, alpha-beta pruning 에서의 문제인지 조금씩의 결함이 보였다. 따라서 결과적으로 우리가 선택한 것은 최대한의 depth 를 탐색하기 위해 evaluation 에 소요되는 시간을 최소한으로 단축시키는 것이었고, 그 결과로 현재의 evaluation function 을 구현하게 되었다.
(4) evaluation 을 구현한 후 초기에 몇 가지 문제가 있었는데, 가장 큰 문제는 초반에 몇 번의 수를 비효율적으로

둔다는 것이었다. 처음 이 문제를 발견했을 때에는 단순히 코딩 과정에서 문제가 발생했다고 생각했는데, 디버깅을 통해 확인해보니 우리가 설정한 평가함수(evaluation function)에서는 실제로 첫 수를 2 에 두는 것이 3 이나 5 보다 큰 score 를 가졌기 때문에 2 에 첫 수를 두는 것이었다. 따라서 우리는 초반 7 수 까지를 최적의 솔루션으로 미리 저장해 놓고 불러 사용하는 방식을 구현하였다. 이는 결과적으로 전체의 경우의 수를 최대 1/7^3 로 줄이는 결과가 되어 정확도 향상에 기여하였다.

(5) 그 외에도 42 비트의 이진수로 보드를 구현하는 방식으로 인해 보드의 경계 구분에서 문제가 발생하기도 했으면 (결과적으로 mask 를 추가해 해결할 수 있었다.) checkwinposition(한 수만에 이길 수 있는 수를 찾는 함수)를 사용하는 위치를 잘못 설정해 문제가 되거나 alpha-beta pruning 에서 prun 조건을 잘못 설정해 문제가 되는 경우도 있었다.

3.3. 감상 및 총평.

(1) 아쉬운 점 :

evaluation 의 정밀도와 시간복잡도 사이의 tradeoff 에 대한 고민이 큰 아쉬움으로 남았다. evaluation function 은 필연적으로 모든 말단 노드에서 수행되어야 하기 때문에, 사실상 전체 프로그램의 시간복잡도를 그대로 n 배가 되게 한다. 따라서 evaluation 의 시간복잡도를 낮추는 것은 결국 search 함수의 depth 를 높일 수 있는 가능성을 의미한다. 하지만 depth 를 하나 또는 둘 높이는 것이 정말 크게 다른 결과를 보이는 지는 알 수 없었다. 어떤 경우에는 낮은 depth 가 옳은 선택을 하는 경우도 있었기 때문이다. 그렇다고 evaluation 의 정확도를 높이기 위해 많은 시간을 소모하는 것도 과연 옳은 일인지 알 수 없었다. evaluation function 을 evaluate 하는 정확한 방법이 없고, 완벽한 평가함수가 존재하는 지의 여부도 찾지 못했기 때문이다. 결과적으로는 evaluation function 의 시간복잡도를 최소한으로 줄이고 depth 를 최대한으로 확보하는 방향으로 프로그램을 설계했지만, depth 를 13 으로 설정할 경우 약간의 시간 초과가 발생해 12 로 설정할 수 밖에 없었다. 그래서 다음에 비슷한 문제를 해결할 때에는 evaluation 을 더 정밀하게 해보아야겠다는 생각이 들었다.

(2) 느낀 점 :

팀프로젝트를 진행하는 것은 전반적으로 재미있었다. 초기 alpha-beta pruning 에 대한 개념 학습도 부족했던 때에는 막연한 두려움이 있었으나, 어느정도 이론에 익숙해지고 난 이후에는 어떤식으로 구현해야 할지를 머리속으로 구상할 수 있었다. 실제로 프로그램을 구현하고 일일히 수작업으로 master 배열을 작성하는 것도 일종의 보람을 느꼈지만, 무엇보다도 승패가 존재하는 게임이었기에 승리를 위한 다양한 전략을 구상하고 토론하는 과정에서 배운 것이 더 많다고 생각한다. 미완성된 프로그램으로 다양한 다른 프로그램들과 대결을 벌이면서 패배의 쓴맛을 맛보고 승리의 짜릿함을 느꼈으며, 여러번의 테스트를 거치며 수정 및 보완하는 작업을 반복하며 일종의 승부욕도 가질 수 있었다.

(3) 역할 분담 : 전반적인 프로그램에 대한 전략이나 구상은 모두가 같이 했다. 함께 의논한 내용을 실제 코딩을 통해 구현하는 과정과 보고서의 결론부분은 박진호가 담당했으며, 마스터 솔루션의 값을 채우고 정리하거나 보고서에 들어갈 자료 및 그림 제작은 김태현이 담당했다. 보고서의 알고리즘 이론 설명 및 본론의 소스에 대한 분석은 이세임이 작성하였다.