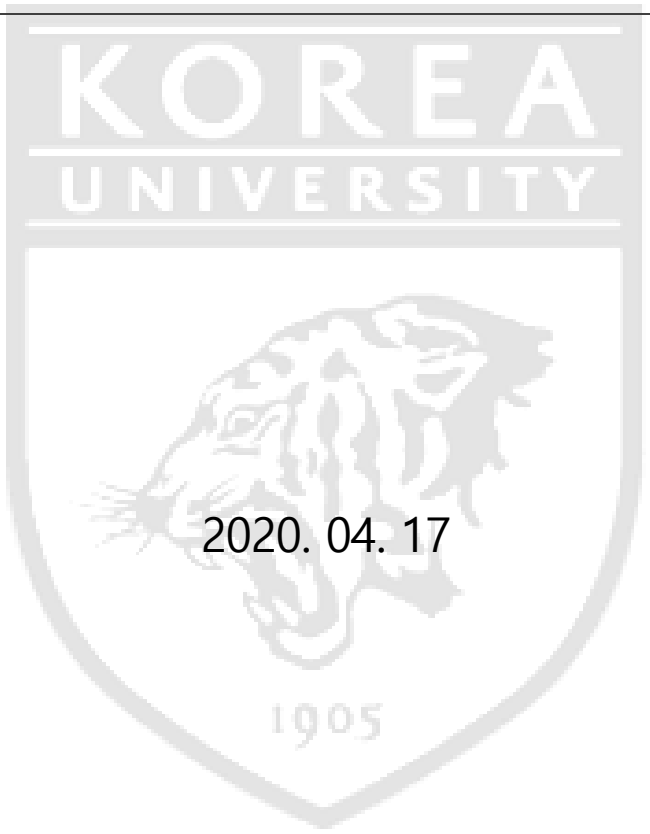
Implement ‘Gomoku’ Game-

Alpha-Beta Search with Iterative Deepening Search



2020. 04. 17

컴퓨터학과

2018320241 정소희

목차

구현 환경

C++ 을 사용해 게임을 구현하였다.

ubuntu 18.04 LTS 에서 g++ command 를 사용해 컴파일하고 게임을 실행하였다.

프로그램 설계

GUI 를 구현하지 않고 AI 와 사용자가 각각 돌을 놓을 때 마다 바뀐 state 상태를 화면에 19X19 크기의 grid 로 출력하였다.

시작할 때 사용자와 AI 의 돌 색깔은 random 하게 결정된다.

사용자와 AI 모두 3-3 이 되는 자리에는 놓을 수 없다. 아래 그림을 참고하여 J 자리에는 놓을 수 없도록 구현하였다.

1. state

state 는 19X19 grid 에 돌들이 어떻게 놓여있는지 저장해 놓은 1 차원 배열

char cur_state[400] 로 설정하였다. 오목판의 각 361 개 자리는 각각 0 번~360 번 위치로써, cur_state 배열의 index 는 자리의 번호가 된다. 초기 state 에는 각 자리에 '-'가 저장되어 있다. '-'는 빈 자리를 나타낸다.

2. action

action 은 오목판 위에 돌을 놓는 행위로, 행위자의 돌 색깔 정보를 cur_state 의 적절한 위치에 저장하는 행위와 같다. 이 때 그 위치는 action 으로, action 이 valid 하면 cur_state[action]에 행위자의 돌 색깔이 저장된다. action 을 통해 state 의 transition 이 일어나게 된다.

3. time limit

AI 와 Player 모두 돌을 놓을 수 있는 시간을 30 초로 제한하였다. 실제로 타자를 치고 오타를 수정하는 과정을 모두 고려하여 넉넉하게 30 초로 설정하였다. 30 초가 지나서 돌을 놓으면 놓아진 돌은 회수되고 게임에서 지게 된다. AI 와 Player 는 모두 시간이 초과되어도 일단 돌을 놓을 수 있지만 바로 시간이 초과됨이 알려지고 돌이 회수되면서 게임이 종료된다. time limit 은 time.h 의 clock_t, CLOCKS_PER_SEC 를 사용해 측정하였다.

```
// example : 시간 측정 예시
 clock_t start_time,search_time;
 start_time=clock();
        ---작업 수행---
search_time=(clock()-start_time)/CLOCKS_PER_SEC;
```

함수 설명

1. main()

main 함수에서 cur_state[400] 배열의 모든 원소를 '-'로 초기화한다.

```
for(int i=0;i<361;i++){
        cur_state[i]='-';
}
```

그리고 start(cur_state)함수를 호출해 게임을 시작하고, player 가 첫 수를 두도록 한다.

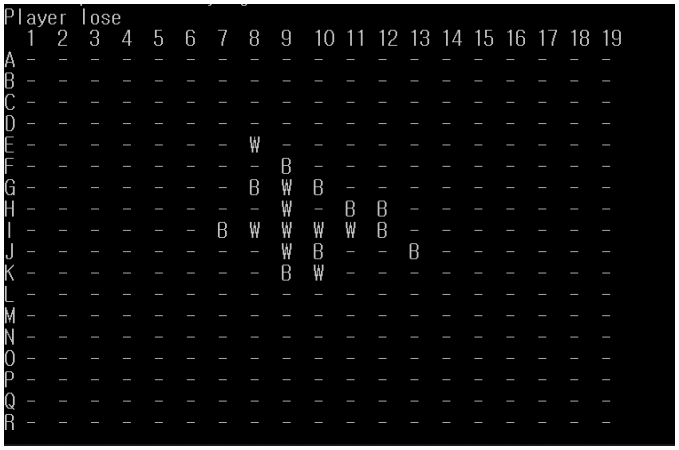
player 가 첫 수를 두고 start 함수를 빠져나오면, 게임 종료 조건을 만족할 때까지 AI 와 player 가 번갈아가며 반복적으로 수를 둔다.

2. 기본적인 기능

a) show(현재 state)

show(cur_state) 함수는 현재 state 가 저장된 1 차원 배열 cur_state 를 for 문을 적절히 사용해 아래와 같이 19X19 grid 로 보여준다. 행은 A,B,...,S, 열은 1,2,...,19 가 표시되도록 한다.

//example: 경기 진행 상황을 show(state) 한 결과



b) player_turn(현재 state)

player_turn 함수는 player 가 돌을 놓을 차례가 올 때 마다 호출된다. player 가 지정된 형식과 다르게 입력할 경우 다시 입력을 받는다.

player 로부터 x(영문 대문자)와 y(숫자)좌표를 입력 받으면, 입력 받은 좌표를 토대로

state[19*(x-65)+(y-1)]=player_color;

와 같이 state 의 action 에 해당하는 위치에 player_color 에 해당하는 색깔을 대입함으로써 state transition 을 일으킨다.

3. valid 한 action 인지 테스트

a) action_isvalid (현재 state, action, ai or player)

action_isvalid 함수는 ai 와 player 모두에게 적용되는 함수로, 수행하려고 하는 action 이 valid 한 action 인지 검사한다.

valid 하지 않은 경우는 다음 3 가지로 설정하였다.

1. 이미 그 자리에 돌이 놓여져 있다 : state[action]=='-'(빈칸) 인지 확인한다.

```
if(state[action]!='-') //빈 자리가 아님
    return 0;
```

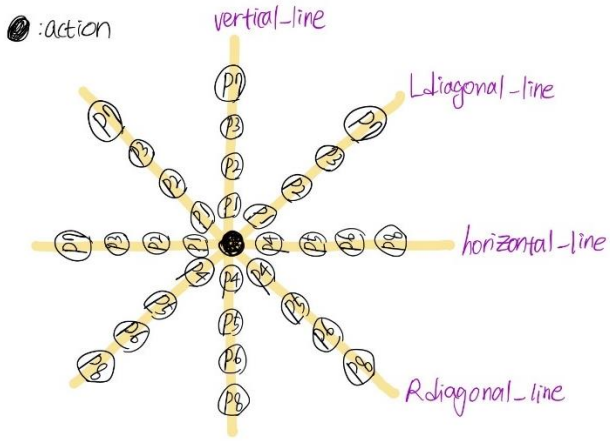
2. 주변 반경 2 칸 이내에 돌이 없다 (너무 동떨어진 위치) : 놓으려고 하는 action 에 해당하는 칸 주변에 돌이 있는지 반복문을 통해 확인한다. 이 제한 조건은 ai 에게만 적용한다. (player 의 돌발 행동은 ai 에게 이득이므로)

3. 그 자리에 돌을 놓으면 3-3 이 만들어진다 : check_33 함수를 통해 확인한다.

b) check_33 (state, color, p1, p2, ..., p8)

check_33 으로 전해지는 인자 중 color 는 검사 대상의 돌의 색깔을 의미한다.

p1, p2, p3, ... p7, p8 는 3-3 을 만드는 위치인지 검사하고자 하는 action 의 주변 위치이다. 그 위치는 아래 그림과 같다.



action_isvalid 내에서 horizontal, vertical, left diagonal, right diagonal 4 개의 line 에 대해서 각각의 p1~p8 값으로 check_33 호출한다.

각 line 에 대해 'action'위치를 포함해서 연속된, 혹은 빈칸이 1 개 존재하는 3 개의 'color' 색깔의 돌이 발견되면 check_33 에서 1 을 return 한다.

action_isvalid 에서는 몇 개의 line 이 1 을 return 받는지 line_count 변수를 통해 확인하고, line_count>=2 이면 3-3 을 만드는 위치인 것으로 판단하여 invalid 하다고 return 한다.

```
//example: action_isvalid 함수 내에서 vertical line check 과정
p1=action-19; p2=action-19*2; p3=action-19*3; p4=action+19; p5=action+19*2; p6=action+19*3; p7=p1-19; p8=p6+19;
if(check_33(state,color,p1,p2,p3,p4,p5,p6,p7,p8)) {
        line_count+=1;
        if(line_count>=2) return 0;
}
```

4. terminal state 인지 테스트

terminal state 는

1. ai 또는 player 가 5 개 연속 돌을 놓았을 때 또는
2. 더 이상 빈 자리가 없을 때 발생한다.

a) terminal_test (state)

2 번 조건은 반복문으로 state 의 모든 자리를 차례로 검사하다가 빈 자리가 존재하면 반복문을 빠져나오는 것을 통해 확인할 수 있다.

1 번 조건은 가로, 세로, 오른쪽 대각선, 왼쪽 대각선 방향으로 연속된 5 개의 같은 색깔의 돌이 존재하는지 검사함으로써 알 수 있다.

```
//모든 자리가 다 찼을 때 -> return 2
int flag=0;
for(int i=0;i<20;i++){
        for(int j=0;j<20;j++){
                if(state[i*19+j]=='-')
```

```
                flag=1; //빈 자리가 존재
            }
        }
        if(!flag)
            return 2;
```

b) 각 방향으로 연속된 돌의 개수를 체크하는 함수들

int horizontal_win(state,color), int vertical_win(state,color),

int Rdiagonal_win(state,color), int Ldiagonal_win(state,color)

은 각각 수평, 수직, 오른쪽 대각선, 왼쪽 대각선 방향으로 최대 몇 개의 같은 색깔 돌이 연속해서 나타나는지 return 한다.

이 함수들은 terminal_test (state)에서 호출되고 다음과 같이 사용된다.

```
        //user 의 우승 -> return -1
if(horizontal_win(state,player_color)==5||vertical_win(state,player_color)==5
||Rdiagonal_win(state,player_color)==5||Ldiagonal_win(state,player_color)==5)
            return -1;
        //ai 의 우승 -> return 1
if(horizontal_win(state,ai_color)==5||vertical_win(state,ai_color)==5||Rdiagonal_win(state,ai_color)==5||Ldiagonal_win(state,ai_color)==5)
            return 1;
```

5. evaluation function

a) estimate

evaluation function 인 estimate(action,state)는 뒤에서 설명할 Alpha_Beta_Search 에서 호출되어 특정 action 이 만드는 state 의 utility 를 계산하여 return 한다.

```
//////////////////////////////////////////////////////////
int estimate(int action,char state[400]){
    int util=0; //default
    int x=action/19; int y=action%19;

    int terminal=five_in_row(state,action);
    if(terminal!=0)
        return terminal;

    //모든 자리가 다 찼을 때 -> return 2
    int flag=0;
    for(int i=0;i<361;i++){
        if(state[i]=='-'){
            flag=1; //빈 자리가 존재
            break;
        }
    }
    if(!flag) return -10; //빈 자리 없음

    util = block_player_5(state,action)+ make_open_4(state,action)+ block_player_4(state,action)
        +block_player_3(state,action)+ block_closed_3(state,action)+ make_open_3(state,action)+ make_open_2(state,action);

    return util;
}
//////////////////////////////////////////////////////////
```

utility 계산에 앞서 만약 terminal state 라면 ai 가 이기는 경우 매우 큰 값으로 정의한 INF*5 를, player 가 이기는 경우 매우 작은 값으로 정의한 NINF*5 를 return 한다. 이 함수는 매우 반복적으로 호출되므로, 여기서 terminal state 인지의 여부는 terminal_test(state) 보다 빠른 five_in_row(state, action)을 사용한다.

utility 를 계산하기 위해 8 가지의 evaluation criteria 를 가지고 8 개의 값을 계산하여 서로 다른 가중치를 부여해 더하고, 그 결과를 return 한다. 8 가지 criteria 에 해당하는 값을 계산하는 8 개의 함수들은 아래와 같다. 아래 8 개의 함수들은 주어진 'action' 을 중심으로 그 주변에 대해서만 검사하는 함수들로, 시간복잡도가 낮다. 하지만 위의 check_33 에서 소개한 horizontal, vertical, right diagonal, left diagonal 에 대해 각각 검사해야 하므로 함수의 길이는 다소 긴 편이다. 각 위치에 대해 돌이 있고,없고의 여부를 조건문으로 검사했다.

b.1) five_in_row (state,action)

//horizontal line 예시

```
int five_in_row(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;
    //1)horizontal line

    count=0;
    for(int i=0;i<19;i++){
        for(int j=0;j<19;j++){
            if(state[i*19+j]==player_color){
                count++;
                if(count==5) return NINF*5; //5개연속 -> win,terminate 조건
            }
            else
                count=0;
        }
        count=0;
    }

    for(int i=-4;i<5;i++){
            if(state[x*19+(y+i)]==ai_color){
                count++;
                if(count==5)
                    return INF*5;
            }
            else count=0;
    }
```

: action 을 수행했을 때 player 또는 ai 가 연속된 5 개의 돌을 가지는지 확인한다. 만약 action 수행으로 인해 ai 가 연속된 5 개의 돌을 가지면 INF*5 를 return 한다. 반면 player 가 연속된 5 개의 돌을 가진다면 NINF*5 를 return 한다.

b.2) block_player_5 (state,action)

: action 을 수행했을 때 한 쪽이 막힌 Player 의 연속된 4 개의 돌의 반대편이 action 에 의해 차단되는지 확인한다. player 의 우승을 막는 그러한 action 이라면 5000 을 return 한다.

//horizontal line 예시

```
int block_player_5(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;

    //1)horizontal line
    count=0;
    if(state[x*19+(y+5)]==ai_color){
        for(int i=1;i<5;i++){
            if(state[x*19+(y+i)]==player_color){
                count++;
                if(count==4)
                    return 5000;
            }
            else count=0;
        }
    }
    else if(state[x*19+(y-5)]==ai_color){
        for(int i=1;i<5;i++){
            if(state[x*19+(y-i)]==player_color){
                count++;
                if(count==4)
                    return 5000;
            }
            else count=0;
        }
    }
```

b.3) make_open_4 (state,action)

: action 을 수행했을 때 양 쪽이 뚫린(=양 쪽이 빈칸인) AI 의 연속된 4 개의 돌의 생성되는지 확인한다. AI 의 우승을 가져오는 그러한 action 이라면 1000 을 return 한다.

//horizontal line 예시

```
int make_open_4(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;

    //horizontal line
    for(int i=-4;i<0;i++){
        if(state[x*19+(y+i)]=='-'){ //empty 만나면
            for(int j=1;j<5;j++){ //그 다음거부터 연속 4개 ai_color인지 검사
                if(state[x*19+(y+i+j)]==ai_color){
                    count++;
                    if(count==4 && state[x*19+(y+i+j+1)]=='-') //연속 4개고 다음이 empty
                        return 1000;
                    if(count>4){
                        count=0;
                        break;
                    }
                }
                else {
                    count=0;
                    break;
                }
            }//for
        }//if
    }//for
```

b.4) block_player_4 (state,action)

: action 을 수행했을 때 양 쪽이 뚫린 Player 의 연속된 4 개의 돌의 한쪽면이 차단되는지 확인한다. 결국에는 player 가 우승할 것이지만 꼭 필요한 action 이므로 500 을 return 한다.

//horizontal line 예시

```
int block_player_4(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;
//1)horizontal line
    count=0;
    if(state[x*19+(y+5)]=='-'){
        for(int i=1;i<5;i++){
            if(state[x*19+(y+i)]==player_color){
                count++;
                if(count==4)
                    return 500;
            }
            else count=0;
        }
    }
    else if(state[x*19+(y-5)]=='-'){
        for(int i=1;i<5;i++){
            if(state[x*19+(y-i)]==player_color){
                count++;
                if(count==4)
                    return 500;
            }
            else count=0;
        }
    }
```

b.5) block_player_3 (state,action)

: action 을 수행했을 때 양 쪽이 뚫린 Player 의 연속된 3 개의 돌의 한쪽면이 차단되는지 확인한다. 맞다면 300 을 return 한다.

//horizontal line 예시

```
int block_player_3(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;
    //1)horizontal line
    count=0;
    if(state[x*19+(y+5)]=='-'){
        for(int i=1;i<4;i++){
            if(state[x*19+(y+i)]==player_color){
                count++;
                if(count==3)
                    return 300;
            }
            else count=0;
        }
    }
    else if(state[x*19+(y-5)]=='-'){
        for(int i=1;i<4;i++){
            if(state[x*19+(y-i)]==player_color){
                count++;
                if(count==3)
                    return 300;
            }
            else count=0;
        }
    }
```

b.6) block_closed_3 (state,action)

: action 을 수행했을 때 한 쪽이 막힌 Player 의 연속된 3 개의 돌의 반대편 빈칸이 차단되는지 확인한다. 맞다면 100 을 return 한다.

//horizontal line 예시

```
int block_closed_3(char state[400],int action){
    int count;
    int x=(action)/19; int y=(action)%19;
    //1)horizontal line
    count=0;
    if(state[x*19+(y+5)]==ai_color){
        for(int i=1;i<4;i++){
            if(state[x*19+(y+i)]==player_color){
                count++;
                if(count==3)
                    return 100;
            }
            else count=0;
        }
    }
    else if(state[x*19+(y-5)]==ai_color){
        for(int i=1;i<4;i++){
            if(state[x*19+(y-i)]==player_color){
                count++;
                if(count==3)
                    return 100;
            }
            else count=0;
        }
    }
```

b.7) make_open_3 (state,action)

: action 을 수행했을 때 양 쪽이 뚫린 AI 의 연속된 3 개의 돌의 생성되는지 확인한다. 맞다면 50 을 return 한다.

//horizontal line 예시

```
int make_open_3(char state[400],int action){

    int x=(action)/19; int y=(action)%19;

    //horizontal line
    if(state[x*19+(y-2)]=='-'){
        if(state[x*19+(y-1)]=='-'){
            if(state[x*19+(y+1)]==ai_color&&state[x*19+(y+2)]==ai_color&&state[x*19+(y+3)]=='-') return 50;
        }
        else if(state[x*19+(y-1)]==ai_color){
            if(state[x*19+(y+1)]==ai_color&&state[x*19+(y+2)]=='-') return 50;
        }
    }
    else if(state[x*19+(y-2)]==ai_color){
        if(state[x*19+(y-3)]=='-'&&state[x*19+(y-1)]==ai_color&&state[x*19+(y+1)]=='-') return 50;
    }
```

b.8) make_open_2 (state,action)

: action 을 수행했을 때 양 쪽이 뚫린 AI 의 연속된 2 개의 돌의 생성되는지 확인한다. 맞다면 10 을 return 한다.

//horizontal line 예시

```
int make_open_2(char state[400],int action){
    int x=(action)/19; int y=(action)%19;
    //horizontal line
    if(state[x*19+(y-1)]=='-'){
        if(state[x*19+(y+1)]==ai_color&&state[x*19+(y+2)]=='-'){
            return 10;
        }
    }
    else if(state[x*19+(y-1)]==ai_color){
        if(state[x*19+(y-2)]=='-'&&state[x*19+(y+1)]=='-'){
            return 10;
        }
    }
```

6. Search 함수

a) Iterative_Deepening_Search (state)

```
////////////////////////////////////////////////////////
int Iterative_Deepening_Search(char state[400]){
    int action;
    clock_t start_time,search_time;
    start_time=clock();

    for(int depth=1;depth<5;depth++){
        int cutoff=0;

        action=Find_action(state,depth,&cutoff);
        search_time=(clock()-start_time)/CLOCKS_PER_SEC;
        if(search_time>30){
            cout<<"\n\nAI Elapsed time : "<<search_time<<" seconds... \n";
            return -100; //시간 초과
        }

        if(cutoff==0) //cutoff not occured
            return action;
    }
    return action;
}
////////////////////////////////////////////////////////
```

Iterative Deepening Search 는 depth 를 한단계씩 증가시키면서 Find_action 함수를 호출한다. depth 를 증가시킬 때 마다 실행 시간이 30 초를 초과하였는지 확인해 실행 시간이 지나치게 길어지는 것을 방지한다.

Find_action 함수가 depth=0 에 걸려서 return 된 것인지 아니면 terminal state 에 걸려서 return 된 것인지 확인하기 위해 Find_action 에 cutoff 변수의 주소값을 넘겨준다. 만약 depth=0 에 걸린다면 cutoff 의 주소에 접근해 그 값을 1 로 바꾼다.

if(cutoff==0) 을 통해 depth==0 에 걸리지 않고 terminal state 를 통해 return 된 것을 확인하면 Find_action 을 통해 찾은 action 값을 return 한다.

b) Find_action (state,depth, cutoff)

```
////////////////////////////////////////////////////////
int Find_action(char state[400],int depth,int* cutoff){

    int best_value=NINF;
    srand(time(NULL));
    int action = rand() % 360;

    for(int act=0;act<361;act++){
        if(!action_isvalid(state,act,0)) //invalid action
            continue;
        //act에 ai 놨을때 다음엔 player 놔야됨
        state[act]=ai_color;
        int value=Alpha_Beta_Search(act,state,depth-1,NINF,INF,cutoff,1);
        state[act]='-';

        if(value>best_value){
            best_value=value;
            action=act;
        }
    }

    return action;

}
////////////////////////////////////////////////////////
```

Find_action 함수는 Iterative_Deepening_Search 에서 넘겨받은 depth 값으로 Alpha_Beta_Search 를 호출한다.

Find_action 은 현재 상태에서 다음에 수행이 가능한 모든 action 들에 대해, 각각의 Alpha_Beta_Search 결과(value)를 비교하여, 그 값이 가장 큰(best_value) action 을 찾는 함수이다.

for 문 안에서, ai 의 차례(max turn)에서 각 action 을 수행했다고 가정하므로, Alpha_Beta_Search 를 호출할 때 depth 는 1 을 낮추고 player's turn 으로 설정하여 호출한다. (Alpha_Beta_Search(act,state,depth-1,a,b,cutoff,1);)

c) Alpha_Beta_Search (action, state, depth, alpha,beta, cutoff, ai or player)

```
////////////////////////////////////////////////////////
int Alpha_Beta_Search(int action,char state[400],int depth,int a, int b, int* cutoff,int player){

    int est=estimate(action,state); //estimate function
    if(depth==0 || est==INF*5 || est==NINF*5 || est==-10){
        if(!depth){
            (*cutoff)=1; //cutoff occured
        }
        return est;
    }

    if(player==0){ //maximizing(ai playing)
        int value=NINF;
        int new_value;

        for(int act=0;act<361;act++){
            if(!action_isvalid(state,act,0)) //invalid action
                continue;
            (*cutoff)=0;

            state[act]=ai_color;
            new_value=Alpha_Beta_Search(act,state,depth-1,a,b,cutoff,1);
            state[act]='-';

            value=max(new_value,value);
            if(value>=b) return value;
            a=max(a,value);=
        }
        return value;
    }

    else{ //minimizing(player playing)
        int value=INF;
        int new_value;

        for(int act=0;act<361;act++){
            if(!action_isvalid(state,act,0)) //invalid action
                continue;
            (*cutoff)=0;

            state[act]=player_color;
            new_value=Alpha_Beta_Search(act,state,depth-1,a,b,cutoff,0);
            state[act]='-';

            value=min(new_value,value);
            if(value<=a) return value;
            b=min(b,value);

        }
        return value;
    }

}
```

Alpha_Beta_Search 는 Minimax Algorithm 의 효율성을 pruning 을 통해 향상시킨 Algorithm 이다. Alpha_Beta_Search 는 value 를 계산할 대상 action, state 와 depth, alpha,

beta, 그리고 depth=0 에 도달했는지 검사할 cutoff, 마지막으로 ai(maximizing player)인지 player(minimizing player)인지를 입력 받는다.

player 값이 0 인 것은 ai 를, 1 인 것은 사용자(player)를 뜻한다고 정했다.

함수 내에서 if 부분은 max value 를 구하고, else 부분은 min value 를 구한다. max value 와 min value 를 구하는 부분이 Alpha_Beta_Search 를 통해 depth 를 1 씩 감소시키며 서로 반복적으로 호출되다가, depth==0 이거나 terminal state 에 도달하면 estimate function 으로 계산한 value 를 return 한다.

결론(데모 캡처)

데모 영상을 첨부하였습니다.

개선할 부분

Iterative_Deepening_Search 함수에서 depth 제한을 0~4 로 search 할 경우 처음 몇 수는 잘 탐색하지 못하지만 빠르게 진행이 되고 세네 번째 수부터는 제대로 작동한다.
depth 제한을 0~5 이상으로 할 경우 처음 수부터 잘 탐색되지만 각 수 당 1 분 이상으로 긴 시간이 걸린다.

Alpha_Beta_Search 함수가 호출될 때 마다 estimate 함수가 호출되는데, 그 횟수는 depth 가 증가할수록 매우 커진다. 따라서 estimate 함수의 실행 시간을 더 효율적으로 줄이면 전체 search 시간 또한 매우 줄어들 것으로 생각된다.

현재 프로그램으로는 depth<5 로 제한할 경우 처음 1~2 개의 수가 최적의 수가 아닌 것처럼 보이는데, 이것은 estimate 함수의 속도 때문에 depth 를 크게 증가시키지 못하기 때문으로 생각된다.

만약 estimate 의 실행 시간을 줄이게 된다면, depth 가 충분히 커도 적당한 시간 내에 Iterative Deepening Search 가 수행되므로, ai 는 처음 몇 개의 수 또한 더 정확한 예측에 근거해서 오목을 둘 수 있을 것이다.