2022100124\_권서은\_Assignment1

1. **Explanation of the Search algorithm.**

이번 오목게임 프로그래밍과제를 몬테 카를로 트리 서치 알고리즘을 통해서 탐색 알고리즘을 구현하고자 했습니다. 알고리즘 구현을 위해 기본적으로 다음과 같이 정의했습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. createNode :

우선 새로운 노드인 newNode를 만들기에 앞서, memcpy기능을 사용해 보드 상태를 복사해서 (탐색이 실제 보드에 영향을 미치지 않도록) 할당했습니다. 또한 moveX와 moveY로 이제 보드상 위치를 구현하려 했고, 아직 수행된 이동이 없는 만큼 해당 로직에선 -1로 설정했습니다.

1. selectNode:

먼저 calculateUCB1함수를 통해 특정 노드의 UCB1값을 계산했습니다. UCB1 공식을 사용해 각 노드의 exploitative측면과 explorative측면을 모두 고려했습니다. 이후 selectNode함수에서 현재 노드에 자식노드가 있는 경우에, 자식 노드의 수만큼 반복하며 각 노드의 UCB1값을 계산한 뒤, 가장 높은 UCB1값을 가진 자식 노드를 선택했습니다.

1. expandNode:

이제 트리를 확장하는 expandNode함수는 조금 다양한 함수를 여러 개 사용하며 구현했습니다. 우선 generateNextMove함수를 통해, 비어 있는 (아직 돌이 놓이지 않은) 모든 칸을 이동 가능한 위치라고 파악하고, 가능한 모든 오목판 상 위치를 배열로 반환했습니다. (board[2][3]가 현재 비어있다면, Move { x = 3, y =2}가 반환되었습니다.) 이후 시간 상의 제약을 고려해, 가능한 모든 이동 위치에서 “유효한” 이동 위치를 파악하려 했습니다.

1. 플레이어가 둔 돌 주변 +-3칸 위주로 탐색. 이 경우에 주변에 인간 플레이어가 둔 돌이 있으면 유효한 이동으로 판단, 없다면 유효하지 않은 이동으로 판단했습니다. (isValidMove =true;)
2. 원래 두던 곳과 너무 거리가 멀 경우 (maxDistance = 7) 해당 돌은 무시.
3. 생성되는 childNode의 점수를 계산하여 (이는 claculateScores 함수를 활용했습니다.)점수가 높은 순서대로 자식 노드 확장.

하는 방식으로 효율적으로 현 노드의 자식 노드를 확장하고자 했습니다.

1. simulateNode:

이제 노드의 상태를 기반으로 게임을 시뮬레이션 했습니다. 비어있는 보드판에 돌을 두며 만약 게임을 이겼을 경우 최적의 bestX와 bestY를 해당 위치로 설정했습니다. 이후 다시 보드판을 원래 비어있던 상태로 설정했습니다. 만약 최적의 위치가 발견되지 않는다면, 난수 설정을 통해 bestX와 bestY를 설정했습니다. 그리고 현재 ai가 플레이하고 있기 때문에, 만약 플레이어가 이긴다면 해당 위치 점수를 -1, ai가 이긴다면 해당 위치 점수를 +1, 그리고 빈공간도 없지만 이긴 사람도 없다면 무승부처리로 0을 할당했습니다.

1. backpropagate: 이후 현재 노드부터 루트노드까지 다시 거슬러 올라가며 방문 횟수, 점수를 업데이트 했습니다.
2. Explanation of the heuristic function.

우선 먼저 휴리스틱 기능으로

1. expandNode 시 모든 탐색가능한 경우를 살펴보는 것이 아니라 인간 플레이어가 둔 돌 +-3칸 위주로 탐색해, 불필요한 공간에서 탐색하는 것을 방지했습니다.
2. maxDistance =7로 설정하고, 해당 거리를 벗어나서 플레이어가 뒀을 경우, 유효하지 않은 수로 판단했습니다.
3. findBlockingPositinForThreeInRow함수와 findWinningPositionForThreeInRow함수를 통해 우선 상대방이 연속 3돌을 두거나, 본인이 연속 3돌일 경우에 두면 좋을 위치를 구현했습니다.
4. Explanation of how to compile the source code.

첨부한 코드를 compile하기 위해서 우선 visual studio의 소스 파일에 main.c를 구현해야합니다. 이를 통해 해당 코드를 컴파일할 수 있습니다.

1. Explanation of how to run the program.

디버거를 돌려,

“Set time limit for each move (E.g. 10 for 10 seconds):” 라는 문구가 나오면 시간 설정을 초 단위 없이 숫자만 적으셔야합니다. 만약 8초를 timelimit으로 두고 싶다면, 8 을 입력하시고 엔터를 누르세요.

“Choose the color you want. Type black as B, white as W. Black player starts first:” 컴퓨터와 사람 중 누가 먼저 돌을 둘 지 결정하세요. 먼저 하고 싶다면 흑돌을 선택하시고, B를 입력하고 엔터를 누르세요. 나중에 하고 싶다면 백돌을 선택하시고, W를 입력하고 엔터를 누르세요.

“Player (B)'s turn. Make your move. (E.g. j 10 for (j, 10)):” 이러한 문구가 나오면 이제 돌을 두시면 됩니다. x축은 알파벳 소문자, y축은 숫자입니다. 만일 (j,10) 위치에 돌을 두고 싶다면 j 10 을 입력하고 엔터를 누르세요. 앞서 설정한 시간을 초과하면 난수를 통해 랜덤하게 돌이 임의로 두어집니다.

1. Program Execution.

진행 과정 캡쳐입니다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명 시간과 돌 설정하는 화면입니다.

텍스트, 스크린샷, 스티치이(가) 표시된 사진

자동 생성된 설명 현재 검은돌이 컴퓨터고, 제가 흰 돌인데, 지금 공격하는 모습을 보실 수 있습니다.

텍스트, 스크린샷, 스티치이(가) 표시된 사진

자동 생성된 설명 예시를 위해 플레이한 다른 게임입니다. 제가 흰돌인데, 제 공격을 막은 모습을 보실 수 있습니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명 예시를 위해 플레이한 다른 게임입니다. 제가 흰돌인데, 동떨어진 곳 (b, 16)에 두자 제 공격을 막은 모습을 보실 수 있습니다.

코드는 다음과 같습니다:

#pragma warning(disable : 4996)

#include <stdio.h>

#include <Windows.h>

#include <math.h>

#include <conio.h>

#include <wchar.h>

#include <locale.h>

#include <time.h>

#include <float.h>

#include <stdbool.h>

#include <string.h>

#define BOARD\_SIZE 19

#define C sqrt(2.0) //UCB1 constant

#define BLOCKED\_FOUR\_SCORE 5000 // 상대방의 금수(5개 연속된 돌을 막음)에 부여되는 점수

#define FOUR\_SCORE 1000 // 자신의 금수(5개 연속된 돌)에 부여되는 점수

#define BLOCKED\_THREE\_SCORE 100 // 상대방의 가로나 세로 4개 중 하나만 막음에 부여되는 점수

#define THREE\_SCORE 10 // 자신의 가로나 세로 4개 중 하나를 완성함에 부여되는 점수

#define BLOCKED\_TWO\_SCORE 5 // 상대방의 세로나 가로 3개 중 하나를 막음에 부여되는 점수

#define TWO\_SCORE 1 // 자신의 세로나 가로 3개 중 하나를 완성함에 부여되는 점수

wchar\_t board[BOARD\_SIZE][BOARD\_SIZE];

void omokBoard();

char gameWin(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE]);

int setTimeLimit;

char playerChoice;

char aiChoice;

void gameMove(char playerChoice, char aiChoice, int setTimeLimit);

typedef enum {

REAL\_PLAYER = 1,

AI\_PLAYER = 2

} Player;

struct State {

wchar\_t board[BOARD\_SIZE][BOARD\_SIZE];

Player currentPlayer;

int previousOpponentMoveX;

int previousOpponentMoveY;

} typedef State;

State currentState;

struct Node {

State\* state;

struct Node\* parent; //부모 노드

struct Node\*\* children; //자식 노드

int numChildren; //자식 노드 수

int wins;

int visits;

int moveX; // 노드에서 실행된 마지막 board[y][x] 에서 x정보

int moveY; // 노드에서 실행된 마지막 board[y][x] 에서 y정보

} typedef Node;

struct Move {

int x;

int y;

} typedef Move;

struct BestMove {

int x;

int y;

} typedef BestMove;

Node\* createNode(State state) { //tree 생성

Node\* newNode = (Node\*)malloc(sizeof(Node));

State\* newState = (State\*)malloc(sizeof(State));

memcpy(newState, &state, sizeof(State));

newNode->state = newState;

newNode->moveX = -1; // 아직 수행된 이동이 없음

newNode->moveY = -1; // 아직 수행된 이동이 없음

newNode->wins = 0;

newNode->visits = 0;

newNode->parent = NULL;

newNode->children = NULL;

newNode->numChildren = 0;

return newNode;

}

Node\* selectNode(Node\* node, int prevOpponentMoveX, int prevOpponentMoveY) {

while (node->numChildren > 0) {

double maxUCB1 = -DBL\_MAX;

Node\* bestChild = NULL;

for (int i = 0; i < node->numChildren; i++) {

Node\* child = node->children[i];

double UCB1;

if (child->visits > 0) {

double explorationTerm = C \* sqrt(log(node->visits) / child->visits);

double exploitationTerm = ((double)child->wins / (double)child->visits);

double proximityTerm = 0.0;

if (prevOpponentMoveX != -1 && prevOpponentMoveY != -1) {

double distance = sqrt(pow(child->moveX - prevOpponentMoveX, 2) + pow(child->moveY - prevOpponentMoveY, 2));

proximityTerm = 1.0 / (1.0 + distance); // 필요에 따라 이 요소 조정

}

UCB1 = exploitationTerm + explorationTerm + proximityTerm;

}

else {

UCB1 = DBL\_MAX;

}

if (UCB1 > maxUCB1) {

maxUCB1 = UCB1;

bestChild = child;

}

}

if (bestChild != NULL) {

node = bestChild;

}

else {

break;

}

}

return node;

}

Move\* generateNextMove(Node\* node, int\* numMoves) {

Move\* moves = (Move\*)malloc(BOARD\_SIZE \* BOARD\_SIZE \* sizeof(Move)); //최대 가능한 이동의 수

int count = 0;

for (int y = 0; y < BOARD\_SIZE; y++) {

for (int x = 0; x < BOARD\_SIZE; x++) {

if (node->state->board[y][x] == L'\u253C') {

moves[count].x = x;

moves[count].y = y;

count++;

}

}

}

\*numMoves = count; // 가능한 이동의 수를 설정합니다.

return moves; // 가능한 모든 이동의 배열을 반환합니다.

}

State\* applyMove(Node\* node, Move move, char playerChoice) {

State\* newState = (State\*)malloc(sizeof(State));

for (int y = 0; y < BOARD\_SIZE; y++) {

for (int x = 0; x < BOARD\_SIZE; x++) {

newState->board[y][x] = node->state->board[y][x];

}

} //현재상태의 보드

newState->currentPlayer = node->state->currentPlayer;

wchar\_t playerDol;

wchar\_t aiDol;

playerDol = (playerChoice == 'B') ? L'○' : L'●';

aiDol = (playerChoice == 'B') ? L'●' : L'○';

if (newState->currentPlayer == REAL\_PLAYER) { //이동

newState->board[move.y][move.x] = playerDol;

}

else {

newState->board[move.y][move.x] = aiDol;

}

if (newState->currentPlayer == REAL\_PLAYER) { //플레이어전환

newState->currentPlayer = AI\_PLAYER;

}

else {

newState->currentPlayer = REAL\_PLAYER;

}

return newState;

}

int horizontalCount(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE], int y, int x, wchar\_t playerDol) { //가로

int count = 0;

for (int i = x; i >= 0; i--) {

if (board[y][i] == playerDol) {

count++;

}

else {

break;

}

}

for (int i = x + 1; i < BOARD\_SIZE; i++) {

if (board[y][i] == playerDol) {

count++;

}

else {

break;

}

}

return count;

}

int verticalCount(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE], int y, int x, wchar\_t playerDol) { //세로

int count = 0;

for (int i = y; i >= 0; i--) {

if (board[i][x] == playerDol) {

count++;

}

else {

break;

}

}

for (int i = y + 1; i < BOARD\_SIZE; i++) {

if (board[i][x] == playerDol) {

count++;

}

else {

break;

}

}

return count;

}

int diagonalCount(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE], int y, int x, wchar\_t playerDol) { //대각선

int count = 0;

for (int i = y, j = x; i >= 0 && j >= 0; i--, j--) {

if (board[i][j] == playerDol) {

count++;

}

else {

break;

}

}

for (int i = y + 1, j = x + 1; i < BOARD\_SIZE && j < BOARD\_SIZE; i++, j++) {

if (board[i][j] == playerDol) {

count++;

}

else {

break;

}

}

return count;

}

int antiDiagonalCount(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE], int y, int x, wchar\_t playerDol) { //반대대각선

int count = 0;

for (int i = y, j = x; i >= 0 && j < BOARD\_SIZE; i--, j++) {

if (board[i][j] == playerDol) {

count++;

}

else {

break;

}

}

for (int i = y + 1, j = x - 1; i < BOARD\_SIZE && j >= 0; i++, j--) {

if (board[i][j] == playerDol) {

count++;

}

else {

break;

}

}

return count;

}

void calculateScores(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE], char playerChoice, int scores[BOARD\_SIZE][BOARD\_SIZE]) {

for (int y = 0; y < BOARD\_SIZE; y++) {

for (int x = 0; x < BOARD\_SIZE; x++) {

if (board[y][x] != L'\u253C') {// 빈 칸이 아닌 경우 점수 계산을 수행 안함

scores[y][x] = 0;

continue;

}

int score = 0;

wchar\_t playerDol = (playerChoice == 'B') ? L'○' : L'●';

int horizontalScore = horizontalCount(board, y, x, playerDol); //가로방향점수

int verticalScore = verticalCount(board, y, x, playerDol); //세로방향점수

int diagonalScore = diagonalCount(board, y, x, playerDol); //대각선

int antiDiagonalScore = antiDiagonalCount(board, y, x, playerDol); //반대대각선

if (horizontalScore >= 5 || verticalScore >= 5 || diagonalScore >= 5 || antiDiagonalScore >= 5) { //상대방 5개 연속돌 막기

score += BLOCKED\_FOUR\_SCORE;

}

else if (horizontalScore == 5 || verticalScore == 5 || diagonalScore == 5 || antiDiagonalScore == 5) { //자신 5개 연속돌 두기

score += FOUR\_SCORE;

}

else if (horizontalScore == 4 || verticalScore == 4 || diagonalScore == 4 || antiDiagonalScore == 4) { //상대방 4개 중 하나만 막기

}

else if (horizontalScore >= 3 || verticalScore >= 3 || diagonalScore >= 3 || antiDiagonalScore >= 3) { //자신 4개 중 하나 완성

score += THREE\_SCORE;

}

else if (horizontalScore == 3 || verticalScore == 3 || diagonalScore == 3 || antiDiagonalScore == 3) { //상대방 3개 중 하나만 막기

score += BLOCKED\_TWO\_SCORE;

}

else if (horizontalScore >= 2 || verticalScore >= 2 || diagonalScore >= 2 || antiDiagonalScore >= 2) { //자식 3개 중 하나 완성

score += TWO\_SCORE;

}

scores[y][x] = score;

}

}

}

void expandNode(Node\* node, char playerChoice) {

wchar\_t playerDol = (playerChoice == 'B') ? L'○' : L'●';

int numMoves;

Move\* possibleMoves = generateNextMove(node, &numMoves);

node->children = (Node\*\*)malloc(numMoves \* sizeof(Node\*));

int numValidMoves = 0;

for (int i = 0; i < numMoves; i++) { //현재 플레이어가 둔 돌 주변을 탐색

bool isValidMove = false;

int moveX = possibleMoves[i].x;

int moveY = possibleMoves[i].y;

int distanceToPrevMoveX = abs(moveX - node->state->previousOpponentMoveX);

int distanceToPrevMoveY = abs(moveY - node->state->previousOpponentMoveY);

int maxDistance = 7; // 최대 허용 거리

if (distanceToPrevMoveX <= maxDistance && distanceToPrevMoveY <= maxDistance) {// 이전 단계와의 거리가 허용 범위 이내에 있는지 확인

for (int dy = -3; dy <= 3; dy++) { // 플레이어가 둔 돌 주변을 탐색

for (int dx = -3; dx <= 3; dx++) {

int x = moveX + dx;

int y = moveY + dy;

if (x >= 0 && x < BOARD\_SIZE && y >= 0 && y < BOARD\_SIZE && node->state->board[y][x] == playerDol) {

isValidMove = true;

}

}

}

}

if (isValidMove) {

Node\* child = (Node\*)malloc(sizeof(Node));

State\* newState = applyMove(node, possibleMoves[i], playerChoice);

child->state = newState;

child->parent = node;

child->children = NULL;

child->numChildren = 0;

child->wins = 0;

child->visits = 0;

child->moveX = moveX;

child->moveY = moveY;

int scores[BOARD\_SIZE][BOARD\_SIZE];

calculateScores(newState->board, playerChoice, scores);

// 점수가 높은 순서대로 자식 노드를 확장

Node\* bestChild = selectNode(child, newState->previousOpponentMoveX, newState->previousOpponentMoveY);

node->children[numValidMoves] = bestChild;

numValidMoves++;

}

}

node->numChildren = numValidMoves; // 실제로 확장된 자식 노드의 수를 업데이트

free(possibleMoves);

}

bool isBoardFull(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE]) {

for (int i = 0; i < BOARD\_SIZE; i++) {

for (int j = 0; j < BOARD\_SIZE; j++) {

if (board[i][j] == L'\u253C') {

return false;

}

}

}

return true;

}

double simulateNode(Node\* node, char playerChoice) { //simulation

State\* currentState = node->state;

wchar\_t playerDol = (playerChoice == 'B') ? L'○' : L'●';

wchar\_t aiDol = (playerChoice == 'B') ? L'●' : L'○';

while (!isBoardFull(currentState->board)) { //보드가 다 차 있지 않다면

int bestX = -1;

int bestY = -1;

for (int y = 9; y < BOARD\_SIZE; ++y) {

for (int x = 9; x < BOARD\_SIZE; ++x) {

if (currentState->board[y][x] == L'\u253c') {

currentState->board[y][x] = (currentState->currentPlayer == REAL\_PLAYER) ? playerDol : aiDol;

if (gameWin(currentState->board) == playerChoice) {

bestX = x;

bestY = y;

}

currentState->board[y][x] = L'\u253c';

}

}

}

if (bestX == -1) {

do {

bestX = rand() % BOARD\_SIZE;

bestY = rand() % BOARD\_SIZE;

} while (currentState->board[bestY][bestX] != L'\u253c');

}

currentState->board[bestY][bestX] = (currentState->currentPlayer == REAL\_PLAYER) ? playerDol : aiDol;

char winner = gameWin(currentState->board);

if (winner != ' ') {

return (winner == playerChoice) ? -1.0 : 1.0;

}

}

// 빈 공간이 없는 경우 무승부로 처리

return 0.0;

}

void backPropagate(double result, Node\* node) { //backPropagation

// 현재 노드부터 루트 노드까지 거슬러 올라가면서 플레이 횟수와 점수를 업데이트

while (node != NULL) {

node->visits++;

node->wins += result;

node = node->parent;

}

}

BestMove selectMostPlayedNode(Node\* node) {

BestMove bestMove = { .x = -1, .y = -1 }; // 기본값으로 9 설정

if (node->numChildren == 0) {

return bestMove; // 자식 노드가 없으면 기본값 반환

}

int maxPlays = -1; // 최대 플레이 횟수를 추적하기 위한 변수

for (int i = 0; i < node->numChildren; i++) {

Node\* childNode = node->children[i];

if (childNode->visits > maxPlays) {

maxPlays = childNode->visits;

bestMove.x = childNode->moveX;

bestMove.y = childNode->moveY;

}

}

return bestMove;

}

bool findBlockingPositionForThreeInRow(Node\* node, Move\* blockMove, wchar\_t playerDol) {

for (int y = 0; y < BOARD\_SIZE; y++) {

for (int x = 0; x < BOARD\_SIZE - 3; x++) { // 가로 방향 연속 확인

int count = 0;

for (int i = 0; i < 3; i++) {

if (node->state->board[y][x + i] == playerDol) {

count++;

}

else if (node->state->board[y][x + i] != L'\u253C') { // 빈 칸이 아니라면

break; // 다른 돌이 있으므로 이 위치에서는 연속된 3개가 될 수 없음

}

}

if (count == 3) {

// 연속된 3개의 돌을 찾았으며, 양쪽 끝이 빈 칸인지 확인

if (x > 0 && node->state->board[y][x - 1] == L'\u253C') {

blockMove->x = x - 1;

blockMove->y = y;

return true;

}

else if (x + 3 < BOARD\_SIZE && node->state->board[y][x + 3] == L'\u253C') {

blockMove->x = x + 3;

blockMove->y = y;

return true;

}

}

}

}

for (int x = 0; x < BOARD\_SIZE; x++) { // 세로 방향 확인

for (int y = 0; y < BOARD\_SIZE - 3; y++) { // 세로 방향 연속 확인

int count = 0;

for (int i = 0; i < 3; i++) {

if (node->state->board[y + i][x] == playerDol) {

count++;

}

else if (node->state->board[y + i][x] != L'\u253C') { // 빈 칸이 아니라면

break; // 다른 돌이 있으므로 이 위치에서는 연속된 3개가 될 수 없음

}

}

if (count == 3) {

// 연속된 3개의 돌을 찾았으며, 위, 아래가 빈 칸인지 확인

if (y > 0 && node->state->board[y - 1][x] == L'\u253C') {

blockMove->x = x;

blockMove->y = y - 1;

return true;

}

else if (y + 3 < BOARD\_SIZE && node->state->board[y + 3][x] == L'\u253C') {

blockMove->x = x;

blockMove->y = y + 3;

return true;

}

}

}

}

for (int x = 0; x < BOARD\_SIZE - 2; x++) { // 우상향 대각선 방향 확인

for (int y = 2; y < BOARD\_SIZE; y++) { // 우상향 대각선 연속 확인

int count = 0;

for (int i = 0; i < 3; i++) {

if (node->state->board[y - i][x + i] == playerDol) {

count++;

}

else if (node->state->board[y - i][x + i] != L'\u253C') { // 빈 칸이 아니라면

break; // 다른 돌이 있으므로 이 위치에서는 연속된 3개가 될 수 없음

}

}

if (count == 3) {

// 연속된 3개의 돌을 찾았으며, 양쪽 끝이 빈 칸인지 확인

if (y + 1 < BOARD\_SIZE && x > 0 && node->state->board[y + 1][x - 1] == L'\u253C') {

blockMove->x = x - 1;

blockMove->y = y + 1;

return true;

}

else if (y - 3 >= 0 && x + 3 < BOARD\_SIZE && node->state->board[y - 3][x + 3] == L'\u253C') {

blockMove->x = x + 3;

blockMove->y = y - 3;

return true;

}

}

}

}

for (int x = 2; x < BOARD\_SIZE; x++) { // 좌상향 대각선 방향 확인

for (int y = 2; y < BOARD\_SIZE; y++) { // 좌상향 대각선 연속 확인

int count = 0;

for (int i = 0; i < 3; i++) {

if (node->state->board[y - i][x - i] == playerDol) {

count++;

}

else if (node->state->board[y - i][x - i] != L'\u253C') { // 빈 칸이 아니라면

break; // 다른 돌이 있으므로 이 위치에서는 연속된 3개가 될 수 없음

}

}

if (count == 3) {

// 연속된 3개의 돌을 찾았으며, 양쪽 끝이 빈 칸인지 확인

if (y + 1 < BOARD\_SIZE && x + 1 < BOARD\_SIZE && node->state->board[y + 1][x + 1] == L'\u253C') {

blockMove->x = x + 1;

blockMove->y = y + 1;

return true;

}

else if (y - 3 >= 0 && x - 3 >= 0 && node->state->board[y - 3][x - 3] == L'\u253C') {

blockMove->x = x - 3;

blockMove->y = y - 3;

return true;

}

}

}

}

return false; // 차단할 위치를 찾지 못함

}

bool findWinningPositionForThreeInRow(Node\* node, Move\* winningMove, wchar\_t aiDol) {

for (int y = 0; y < BOARD\_SIZE; y++) { // 가로 방향으로 연속된 돌이 3개 있는지 확인

for (int x = 0; x <= BOARD\_SIZE - 4; x++) { // 가로 방향 확인

int count = 0; // 연속된 돌의 개수를 세는 변수

for (int i = 0; i < 4; i++) {

if (node->state->board[y][x + i] == aiDol) {

count++;

}

else {

break; // 연속되지 않으면 바로 다음 위치 확인

}

}

if (count >= 3) {

// 연속된 세 개의 돌을 찾음. 이제 양쪽 끝을 확인하여 승리할 위치 찾기

if (x > 0 && node->state->board[y][x - 1] == L'\u253C') { // 왼쪽에 빈 칸이 있는 경우

winningMove->x = x - 1;

winningMove->y = y;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

if (x + 3 < BOARD\_SIZE && node->state->board[y][x + 3] == L'\u253C') { // 오른쪽에 빈 칸이 있는 경우

winningMove->x = x + 3;

winningMove->y = y;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

}

}

}

for (int x = 0; x < BOARD\_SIZE; x++) { // 세로 방향으로 연속된 돌이 3개 있는지 확인

for (int y = 0; y <= BOARD\_SIZE - 4; y++) { // 세로 방향 확인

int count = 0; // 연속된 돌의 개수를 세는 변수

for (int i = 0; i < 4; i++) {

if (node->state->board[y + i][x] == aiDol) {

count++;

}

else {

break; // 연속되지 않으면 바로 다음 위치 확인

}

}

if (count >= 3) {

// 연속된 세 개의 돌을 찾음. 이제 위쪽 또는 아래쪽을 확인하여 승리할 위치 찾기

if (y > 0 && node->state->board[y - 1][x] == L'\u253C') { // 위쪽에 빈 칸이 있는 경우

winningMove->x = x;

winningMove->y = y - 1;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

if (y + 3 < BOARD\_SIZE && node->state->board[y + 3][x] == L'\u253C') { // 아래쪽에 빈 칸이 있는 경우

winningMove->x = x;

winningMove->y = y + 3;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

}

}

}

for (int y = BOARD\_SIZE - 1; y >= 3; y--) { // 보드의 아래쪽에서 시작 // 우상향 대각선 방향으로 연속된 돌이 3개 있는지 확인

for (int x = 0; x <= BOARD\_SIZE - 4; x++) { // 왼쪽에서 시작

int count = 0; // 연속된 돌의 개수를 세는 변수

for (int i = 0; i < 4; i++) {

if (node->state->board[y - i][x + i] == aiDol) {

count++;

}

else {

break; // 연속되지 않으면 바로 다음 위치 확인

}

}

if (count >= 3) {

// 연속된 세 개의 돌을 찾음. 이제 확장할 수 있는 위치 찾기

if (y < BOARD\_SIZE - 1 && x > 0 && node->state->board[y + 1][x - 1] == L'\u253C') { // 왼쪽 아래에 빈 칸이 있는 경우

winningMove->x = x - 1;

winningMove->y = y + 1;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

if (y - 3 >= 0 && x + 3 < BOARD\_SIZE && node->state->board[y - 3][x + 3] == L'\u253C') { // 오른쪽 위에 빈 칸이 있는 경우

winningMove->x = x + 3;

winningMove->y = y - 3;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

}

}

}

for (int y = BOARD\_SIZE - 1; y >= 3; y--) { // 보드의 아래쪽에서 시작 // 좌상향 대각선 방향으로 연속된 돌이 3개 있는지 확인

for (int x = BOARD\_SIZE - 1; x >= 3; x--) { // 오른쪽에서 시작

int count = 0; // 연속된 돌의 개수를 세는 변수

for (int i = 0; i < 4; i++) {

if (node->state->board[y - i][x - i] == aiDol) {

count++;

}

else {

break; // 연속되지 않으면 바로 다음 위치 확인

}

}

if (count >= 3) {

// 연속된 세 개의 돌을 찾음. 이제 확장할 수 있는 위치 찾기

if (y < BOARD\_SIZE - 1 && x < BOARD\_SIZE - 1 && node->state->board[y + 1][x + 1] == L'\u253C') { // 오른쪽 아래에 빈 칸이 있는 경우

winningMove->x = x + 1;

winningMove->y = y + 1;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

if (y - 3 >= 0 && x - 3 >= 0 && node->state->board[y - 3][x - 3] == L'\u253C') { // 왼쪽 위에 빈 칸이 있는 경우

winningMove->x = x - 3;

winningMove->y = y - 3;

return true; // 승리할 수 있는 위치를 찾았으므로 true 반환

}

}

}

}

return false; // 승리할 수 있는 위치를 찾지 못함

}

BestMove mctSearch(State state, int setTimeLimit, char playerChoice, int iterations) {

clock\_t startTime = clock(); // 시작 시간 측정

Node\* tree = createNode(state);

int timeLimit = setTimeLimit \* 1000; // clock() 반환값에 맞게 시간 제한 조정

for (int i = 0; (i < iterations) && ((clock() - startTime) < timeLimit); i++) {

Node\* leaf = selectNode(tree, tree->state->previousOpponentMoveX, tree->state->previousOpponentMoveY); // 이전 상대의 이동 좌표 전달

expandNode(leaf, playerChoice); // 노드 확장

Node\* child = selectNode(leaf, leaf->state->previousOpponentMoveX, leaf->state->previousOpponentMoveY); // 확장된 노드 중 하나 선택

double result = simulateNode(child, playerChoice); // 시뮬레이션 실행

backPropagate(result, child); // 결과 역전파

}

Move blockMove;

Move winMove;

wchar\_t playerDol = (playerChoice == 'B') ? L'○' : L'●';

wchar\_t aiDol = (playerChoice == 'B') ? L'●' : L'○';

wchar\_t currentDol = (state.currentPlayer == REAL\_PLAYER) ? playerDol : aiDol;

if (findWinningPositionForThreeInRow(tree, &blockMove, aiDol)) {

BestMove bestMove = { .x = blockMove.x, .y = blockMove.y };

return bestMove;

}

else if (findBlockingPositionForThreeInRow(tree, &winMove, playerDol)) {

BestMove bestMove = { .x = winMove.x, .y = winMove.y };

return bestMove;

}

else {

// 가장 많이 방문된 노드 선택

Node\* mostVisitedNode = tree->children[0]; // 초기화

for (int i = 1; i < tree->numChildren; i++) {

if (tree->children[i]->visits > mostVisitedNode->visits) {

mostVisitedNode = tree->children[i];

}

}

// 선택된 노드의 좌표 반환

BestMove bestMove = { .x = mostVisitedNode->moveX, .y = mostVisitedNode->moveY };

return bestMove;

}

}

int main() {

setlocale(LC\_ALL, "");

printf("\n OMOK GAME!\n"); //title

for (int i = 0; i < BOARD\_SIZE; i++) {

for (int j = 0; j < BOARD\_SIZE; j++) {

currentState.board[i][j] = L'\u253C'; // '┼'의 유니코드 코드 포인트

}

}

omokBoard();

printf("------------------------------------------\n");

printf("Set time limit for each move (E.g. 10 for 10 seconds):\n");

int scanfResult = scanf("%d", &setTimeLimit);

while (scanfResult != 1 || setTimeLimit < 1) {

printf("Invalid time limit. Time limit should be greater than 1 second.\n");

while (getchar() != '\n');

printf("Set time limit:\n");

scanfResult = scanf("%d", &setTimeLimit);

}

printf("Time limit set to: %d seconds\n", setTimeLimit);

printf("Choose the color you want. Type black as B, white as W. Black player starts first:\n");

scanf(" %c", &playerChoice); //흑돌 백돌 정하기

while (playerChoice != 'B' && playerChoice != 'W') {

printf("Invalid input. Choose between B and W.\n");

printf("Choose the color you want. Type black as B, white as W. Black player starts first:\n");

scanf(" %c", &playerChoice);

}

aiChoice = (playerChoice == 'B') ? 'W' : 'B';

currentState.currentPlayer = (playerChoice == 'B') ? 1 : 2; //플레이어가 흑돌이라면 currentPlayer = 1, ai가 흑돌이라면 currentPlayer = 2

gameMove(playerChoice, aiChoice, setTimeLimit);

return 0;

}

void omokBoard() { // 오목판 배열 생성 및 초기화 및 출력

int i, j;

printf(" ");

for (i = 0; i < BOARD\_SIZE; i++) {

printf("%c ", 'a' + i);

}

printf("\n");

for (i = 0; i < BOARD\_SIZE; i++) {

printf("%02d ", i + 1);

for (j = 0; j < BOARD\_SIZE; j++) {

wprintf(L"%lc\u2500", currentState.board[i][j]);

}

printf("\n");

}

}

bool isBoardEmpty(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE]) {

for (int i = 0; i < BOARD\_SIZE; i++) {

for (int j = 0; j < BOARD\_SIZE; j++) {

if (board[i][j] != L'\u253C') {

return false; // 보드에 돌이 하나라도 있는 경우 false 반환

}

}

}

return true; // 모든 칸이 비어있는 경우 true 반환

}

void gameMove(char playerChoice, char aiChoice, int setTimeLimit) {

//printf("------------------------------------------\n");

//currentState.currentPlayer = (playerChoice == 'B') ? 1 : 2;

int keyin; //좌표 입력 여부

int timeLimit = setTimeLimit \* 1000;

char x;

int y;

wchar\_t playerDol = (playerChoice == 'B') ? L'○' : L'●';

wchar\_t aiDol = (playerChoice == 'B') ? L'●' : L'○';

char winner = ' ';

while (winner == ' ') {

if (currentState.currentPlayer == 1) { //player차례

printf("------------------------------------------\n");

printf("Player (%c)'s turn. Make your move. (E.g. j 10 for (j, 10)):\n", playerChoice);

int selectedPosition = 0; //already selected position?

int invalidPosition = 0; //invalid position?

while (1) {

time\_t startTime = clock();

keyin = 1;

do {

time\_t currentTime = clock();

if (difftime(currentTime, startTime) > timeLimit) { //시간이 초과한다면

keyin = 0; //키 입력이 없음

break;

}

} while (!kbhit()); //키가 안 눌렸음

if (keyin == 1) {

scanf(" %c %d", &x, &y); // ascii code로 x y 받아옴

int numX = x - 'a';

y--;

if (numX >= 0 && numX < BOARD\_SIZE && y >= 0 && y < BOARD\_SIZE) {

if (currentState.board[y][numX] == L'\u253C') {

currentState.board[y][numX] = playerDol;

omokBoard();

gameWin(currentState.board);

currentState.previousOpponentMoveX = numX;

currentState.previousOpponentMoveY = y;

break;

}

else {

printf("Already selected position. Select again.\n");

continue;

}

}

else {

printf("Invalid position. Select again.\n");

continue;

}

}

else {

int randX, randY;

do {

randX = rand() % BOARD\_SIZE;

randY = rand() % BOARD\_SIZE;

} while (currentState.board[randY][randX] != L'\u253C');

printf("Time limit exceeded. Position (%c, %d) is randomly selected.\n", 'a' + randX, randY + 1);

currentState.board[randY][randX] = playerDol;

omokBoard();

gameWin(currentState.board);

currentState.previousOpponentMoveX = randX;

currentState.previousOpponentMoveY = randY;

break;

}

}

currentState.currentPlayer = 2;

}

else { //ai차례

printf("------------------------------------------\n");

printf("AI (%c)'s turn.\n", aiChoice);

srand(time(NULL));

Node\* root = createNode(currentState);

if (isBoardEmpty(currentState.board)) { // 게임이 처음 시작되었는지 확인

int centerX = BOARD\_SIZE / 2;

int centerY = BOARD\_SIZE / 2;

currentState.board[centerY][centerX] = aiDol;

printf("AI selected (%c, %d).\n", 'a' + centerX, centerY + 1);

omokBoard();

currentState.currentPlayer = 1;

}

else {

BestMove bestMove = mctSearch(currentState, setTimeLimit, playerChoice, 1000);

int x = bestMove.x;

int y = bestMove.y;

currentState.board[y][x] = aiDol;

printf("AI selected (%c, %d).\n", 'a' + x, y + 1);

omokBoard();

gameWin(currentState.board);

currentState.currentPlayer = 1;

}

}

winner = gameWin(currentState.board);

}

if (winner != ' ') { // 승자가 결정된 경우 결과 출력

if (winner == playerChoice) {

printf("\nGAME END. PLAYER (%c) won. CONGRATS!\n", winner);

}

else {

printf("\nGAME END. AI (%c) won. Try once more!\n", winner);

}

}

}

char gameWin(wchar\_t board[BOARD\_SIZE][BOARD\_SIZE]) {

char winner = ' '; // 승자를 저장할 변수

for (int i = 0; i < BOARD\_SIZE; i++) { // 가로

for (int j = 0; j < BOARD\_SIZE - 4; j++) {

if (board[i][j] == L'○' && board[i][j + 1] == L'○' && board[i][j + 2] == L'○' && board[i][j + 3] == L'○' && board[i][j + 4] == L'○') {

winner = 'B';

}

if (board[i][j] == L'●' && board[i][j + 1] == L'●' && board[i][j + 2] == L'●' && board[i][j + 3] == L'●' && board[i][j + 4] == L'●') {

winner = 'W';

}

}

}

for (int i = 0; i < BOARD\_SIZE; i++) { // 세로

for (int j = 0; j < BOARD\_SIZE - 4; j++) {

if (board[i][j] == L'○' && board[i + 1][j] == L'○' && board[i + 2][j] == L'○' && board[i + 3][j] == L'○' && board[i + 4][j] == L'○') {

winner = 'B';

}

if (board[i][j] == L'●' && board[i + 1][j] == L'●' && board[i + 2][j] == L'●' && board[i + 3][j] == L'●' && board[i + 4][j] == L'●') {

winner = 'W';

}

}

}

for (int i = 0; i < BOARD\_SIZE - 4; i++) { // 우하향

for (int j = 0; j < BOARD\_SIZE - 4; j++) {

if (board[i][j] == L'○' && board[i + 1][j + 1] == L'○' && board[i + 2][j + 2] == L'○' && board[i + 3][j + 3] == L'○' && board[i + 4][j + 4] == L'○') {

winner = 'B';

}

if (board[i][j] == L'●' && board[i + 1][j + 1] == L'●' && board[i + 2][j + 2] == L'●' && board[i + 3][j + 3] == L'●' && board[i + 4][j + 4] == L'●') {

winner = 'W';

}

}

}

for (int i = 0; i < BOARD\_SIZE - 4; i++) { // 좌하향

for (int j = 0; j < BOARD\_SIZE - 4; j++) {

if (board[i][j + 4] == L'○' && board[i + 1][j + 3] == L'○' && board[i + 2][j + 2] == L'○' && board[i + 3][j + 1] == L'○' && board[i + 4][j] == L'○') {

winner = 'B';

}

if (board[i][j + 4] == L'●' && board[i + 1][j + 3] == L'●' && board[i + 2][j + 2] == L'●' && board[i + 3][j + 1] == L'●' && board[i + 4][j] == L'●') {

winner = 'W';

}

}

}

return winner;

}