Final Project: Gomoku AI

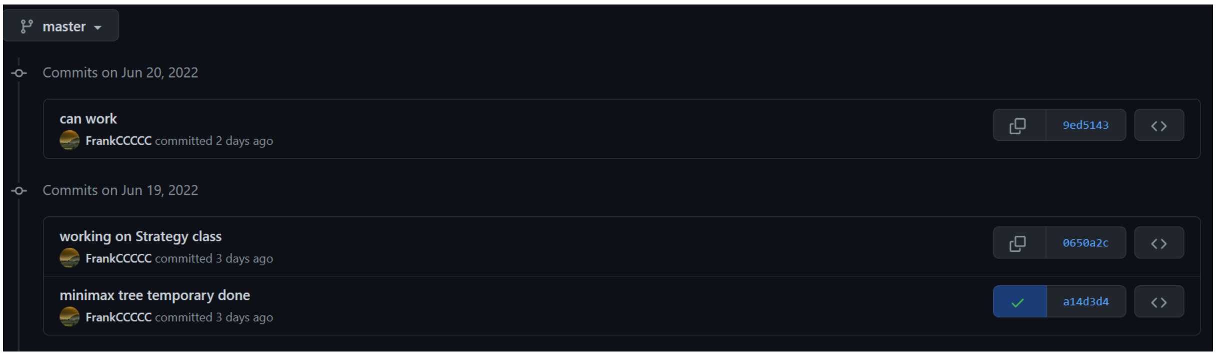
106033233 周聖諺

2022-06-22

Contents

Gomoku AI

Github



Threat Space Search

In some case, if you don' t defense, then you will die unless you can get 5 in a row in 1 move.

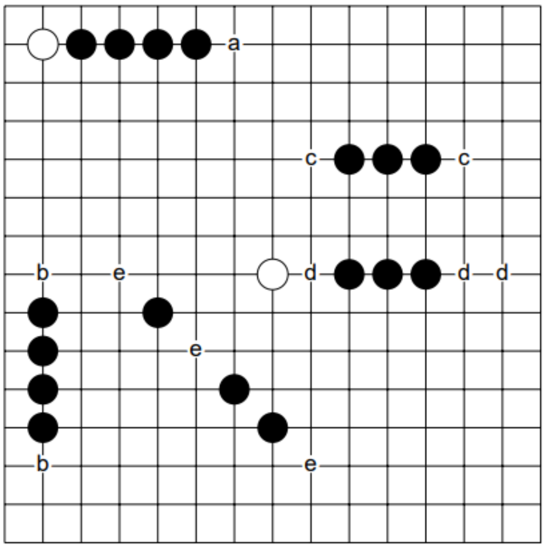


Diagram 1: Threats.

To win the game against any opposition a player needs to create a double threat (either a straight four, or two separate threats). In most cases, a threat sequence, i.e., a series of moves in which each consecutive move contains a threat, is played before a double threat occurs. A threat sequence leading to a (winning) double threat is called a *winning threat sequence*. Each threat in the sequence forces the defender to play a move countering the threat. Hence, the defender's possibilities are limited.

If you want to win, you need to pose double threats. In the ver1, I design a class `state` to record the board, state value and the candidates. But it' s too slow to allocate a string.

Scan opponent' s move to see whether the opponent poses a threat or not. If the opponent poses a threat, search that candidate move first(push the move into the head)

```
bool block_opponent = false;
int tmp_size = std::min(static_cast<int>(moves_opponent.size()), 2);
if (moves_opponent[0].score >= THRAT_SCORE_LIMIT) {
    block_opponent = true;
    for (int i = 0; i < tmp_size; ++i) {
        auto move = moves_opponent[i];

        // Re-evaluate move as current player
        move.score = Eval::eval_pos(state, move.r, move.c, player);

        // Add to candidate list
        candidate_moves.push_back(move);
    }
```

```
}
```

State Value Function

Since we know re-evaluate a state(whole board) is expensive and in evaluation, we actually compute the the same area, I design the state value function that only count the difference of the board, which is the move of the AI and the opponent.

Each move will affect a star area nearby. So, I measure it in 4 directions, which are horizontal, vertical, diagonal directions.

```
void Eval::gen_measures(const char *state, int r, int c, int player,
    bool is_cont, Eval::Measure *ms) {
    ERR_NULL_CHECK(state,)
    ERR_POS_CHECK(r,c,)

    // Scan 4 directions
    gen_measure(state, r, c, Eval::MEASURE_DIR_H, player, is_cont, ms
        [0]);
    gen_measure(state, r, c, Eval::MEASURE_DIR_LU, player, is_cont, ms
        [1]);
    gen_measure(state, r, c, Eval::MEASURE_DIR_V, player, is_cont, ms
        [2]);
    gen_measure(state, r, c, Eval::MEASURE_DIR_RU, player, is_cont, ms
        [3]);
}
```

Each time I record `{Number of pieces in a row, Number of ends blocked by edge or the other player (0-2), Number of spaces in the middle of pattern}`

```
struct Measure {
    // Number of pieces in a row
    char len;
    // Number of ends blocked by edge or the other player (0-2)
    char block_cnt;
    // Number of spaces in the middle of pattern
    char space_cnt;
};
```

And I define the pattern as `{Length of pattern (pieces in a row), Number of ends blocked by edge or the other player (0-2), Number of spaces in the middle of pattern (-1: Ignore value)}`

```
struct Pattern {
    // Minimum number of occurrences to match
    char min_occur;
    // Length of pattern (pieces in a row)
```

```
    char len;
    // Number of ends blocked by edge or the other player (0-2)
    char block_cnt;
    // Number of spaces in the middle of pattern (-1: Ignore value)
    char space_cnt;
};
const Eval::Pattern *Eval::PATTERNS = new Eval::Pattern[PATTERNS_NUM * 2]{
        {1, 5,  0,  0}, {0, 0,  0,  0},  // 10000
        {1, 4,  0,  0}, {0, 0,  0,  0},  // 700
        {2, 4,  1,  0}, {0, 0,  0,  0},  // 700
        {2, 4, -1,  1}, {0, 0,  0,  0},  // 700
        // Threats-----------------------------
        {2, 4,  1,  0}, {0, 0,  0,  0},  // 700
        {2, 4, -1,  1}, {0, 0,  0,  0},  // 700
        {1, 4,  1,  0}, {1, 4, -1,  1},  // 700
        {1, 4,  1,  0}, {1, 3,  0, -1},  // 500
        {1, 4, -1,  1}, {1, 3,  0, -1},  // 500
        {2, 3,  0, -1}, {0, 0,  0,  0},  // 300
        // Threats-----------------------------
        ...
```

How I count the space and the block of a sequence

```
for (int i = 0; i < 2; i++) {
    while (true) {
        // Shift
        r_cnt += dr; c_cnt += dc;

        // Validate position
        if (pos_check(r_cnt, c_cnt)){break;}

    // Get spot value
    int spot = state[_2d_1d(r_cnt, c_cnt)];

    // Empty spots
    if (spot == 0) {
        if (allowed_space > 0 && Util::get_spot(state, r_cnt + dr, c_cnt + dc) == player) {
            allowed_space--;
            res.space_cnt++;
            continue;
        } else {
            res.block_cnt--;
            break;
        }
    }

    // Another player
    if (spot != player){break;}

```

```
    // Current player
    res.len++;
}

// Reverse direction
dr = -dr;
dc = -dc;
r_cnt = r;
c_cnt = c;
}
```

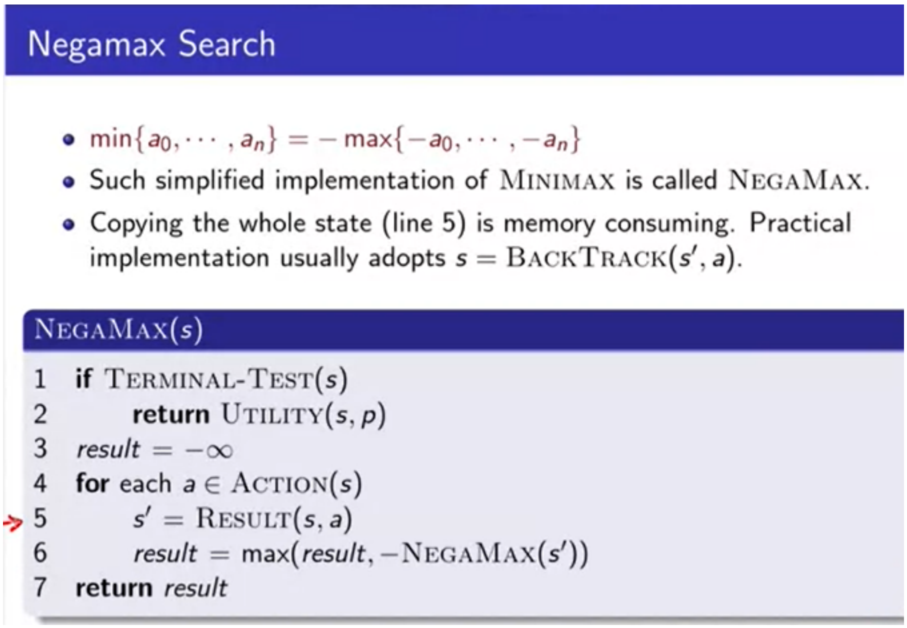
Accumulate the scores along trajectory. Because the `Negamax::negamax` will return opponent's score(score of opponent's utility), we need to minus it.

```
if (depth > 1) sc = negamax(state, opn, initial_depth, depth - 1,
    enable_ab_pruning, -beta, -alpha, dummy_r, dummy_c);

// Decay longer moves
sc = static_cast<int>(sc * SCORE_DECAY);

// Calculate score difference
move.accum_score = move.score - sc;

// Store back to candidate array
cand_mvs.at(i).accum_score = move.accum_score;
```

Minimax & Alpha-Beta Pruning (Negamax)

It's just another implementation of Minimax. It's based on the following formula

$$\max(a, b) = -\min(-a, -b)$$



Pseudo code

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := -∞
    for each child of node do
        value := max(value, -negamax(child, depth - 1, -color))
    return value
```

Negamax with alpha-beta pruning

Pseudo code

```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node

    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := -∞
    foreach child in childNodes do
        value := max(value, -negamax(child, depth - 1, -β, -α, -color))
        α := max(α, value)
        if α ≥ β then
            break (* cut-off *)
    return value
```

```
sc = negamax(state,
```

```
            opponent,           // Change player
            initial_depth,      // Initial depth
            depth - 1,          // Reduce depth by 1
            enable_ab_pruning,  // Alpha-Beta
            -beta,              //
            -alpha,
            move_r,            // Result move
            move_c);

// Store back to candidate array
cand_mvs.at(i).accum_score = move.accum_score;

// Restore the move
Util::set_spot(state, move.r, move.c, 0);

// Update maximum score
if (move.accum_score > max_score) {
    max_score = move.accum_score;
    move_r = move.r; move_c = move.c;
}

// Alpha-beta
if (max_score > alpha) alpha = max_score;
if (enable_ab_pruning && max_score >= beta) break;
```

Iterative Deepening

Re-search the game tree deeper when the time is enough

```
for (int d = INIT_DEPTH;; d += INC_DEPTH) {
    // Reset game state
    memcpy(ng_state, state, G_B_AREA);

    // Execute search
    negamax(ng_state, player, d, d, enable_ab_pruning, alpha, beta,
        move_r, move_c);
    actual_depth = d;
    INFO("Deepening - actual_depth: " << actual_depth << " Act: (" <<
        move_r << ", " << move_c << ")" << " node_count: " << g_node_cnt
         << " eval_count: " << g_eval_cnt)
    io.write_valid_spot(Position(move_r, move_c));
}
```

Performance Issue

- Allocate `std::string`/ append string are very expensive(half of runtime in ver1)
- Allocate class instance is much more expensive

- Use structure, inline function, static method and static variables as much as possible
- Try on ZobristHash to cache the computed states.

```
typedef uint64_t ZbsHash;
typedef uint64_t Hash;

class ZobristHash {
private:
    static ZbsHash* HASH_O, * HASH_X;
public:
    ZobristHash() {}
    static class ClassInit {
        public:
        ClassInit() {
            // Static constructor definition
            std::random_device rd;
            std::mt19937 gen(rd());
            std::uniform_int_distribution<ZbsHash> d(0, UINT64_MAX);

            ZobristHash::HASH_O = new ZbsHash[G_B_AREA];
            ZobristHash::HASH_X = new ZbsHash[G_B_AREA];

            // Generate random values
            for (int i = 0; i < G_B_AREA; i++) {
                ZobristHash::HASH_O[i] = d(gen);
                ZobristHash::HASH_X[i] = d(gen);
            }
        }
    } Initialize;

    static ZbsHash zobrist_hash(const char *state) {
        ZbsHash h = 0;
        for (int i = 0; i < G_B_AREA; i++) {
            if (state[i] == 1) { h ^= HASH_O[i]; }
            else if (state[i] == 2) { h ^= HASH_X[i]; }
        }
        return h;
    }
    static Hash hash(const char *state, int r, int c, int player) {
        Hash h = ZobristHash::zobrist_hash(state);
        h ^= (r ^ c ^ player);
        return h;
    }
};
ZbsHash *ZobristHash::HASH_O, *ZobristHash::HASH_X;
ZobristHash::ClassInit ZobristHash::Initialize;
typedef unordered_map<Hash, Score> STATE_MAP;
```