PROJECT 1

GAME SEARCH STRATEGY

Le Van Hung (黎文雄)

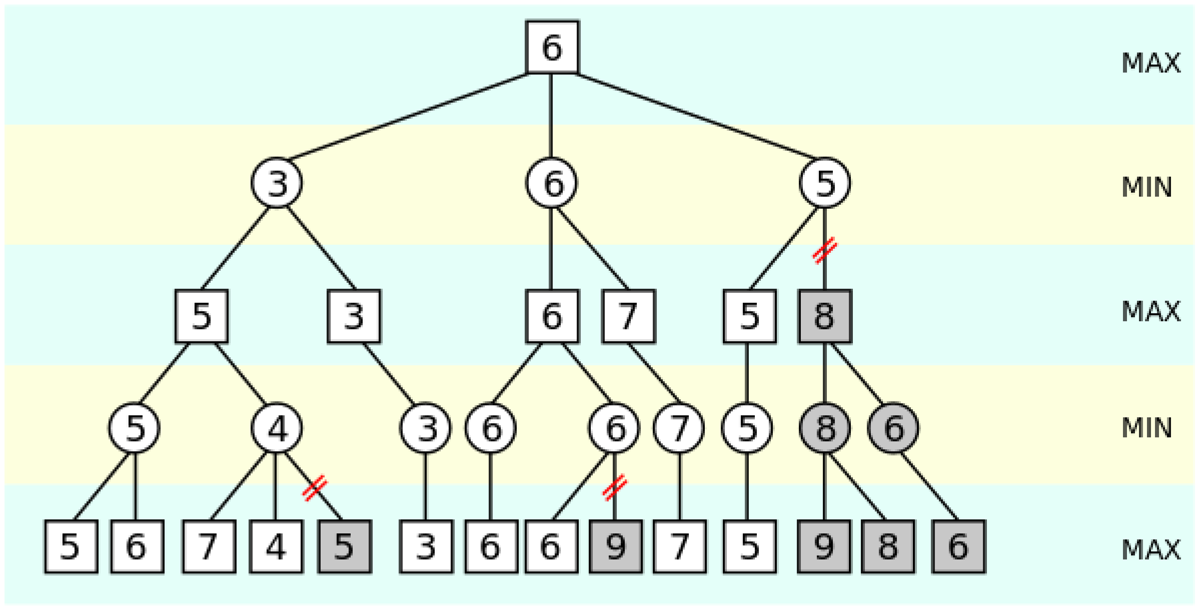
Student ID: 0860831

** Introduction**

In this project, I apply the search strategy based Alpha Beta Pruning Algorithm for searching the final result of this search games.

Because of computation load, I define max_node of Tree, each time with AI turn, base on current board state, I use this to estimate the max of depth of Tree to avoid the out of memory.

1. Alpha beta pruning algorithm



Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tactoe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[1] **(wiki)**

Because of requirement for memory, we can not reach the leaf of the tree(refer in is_terminal() function of class Board), so we need to use this algorithm with depth limited.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

The pseudo-code for depth limited minimax with alpha-beta pruning

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

2. Dynamic depth

Honestly, if we can create the complete tree with all leaves is end-game state, the alpha beta pruning algorithm will always make the AI win. But the lack of memory and runtime, especially in this game with large branching factors (for example: 16*5 branch factors if AI move first in board_size = 4*4) the number of leaves will increase with exponential function.

The idea here is large depth large accuracy. With states near the end-game state, the branching factor will be decrease and we can use larger depth than initial state. Base on the condition of current board, we can decide which depth is best for both accuracy and memory.

3. Set up heuristic function

Evaluation function is used to evaluate a specific state in better or worse than other state by return a score for each state. We have to decide a function that evaluate correctly for each situation of the board.

Because we can not reach the end-game state, so heuristic function is really important to estimate which state will lead to win.

6 factors I used to estimate heuristic function:

- ✓ Sum of card on board of each user: S1
- ✓ Sum of card available of each user: S2
- ✓ Maximum card on board of each user: M1
- ✓ Maximum card available of each user: M2
- ✓ Total number of available cards of each user: N1
- ✓ Total number of deleted cards of each user: N2

Heuristic function will be calculated by the following function:

$$\mathbf{F} = (S1_{ai} + S2_{ai} + M1_{ai} + M2_{ai} + N1_{ai} - 6*N2_{ai})$$

$$- (S1_{user} + S2_{user} + M1_{user} + M2_{user} + N1_{user} - 6*N2_{user})$$

4. Architecture of project

- **main.cpp:** Main file to run project.
- **Board.cpp:**

Define function for class board, contain all the function belong to board and input card.

Each object of class Board consist of **three attributes:** grid[][]: 2D-array to contain board; mark[][] to mark who put that card on the board and size of board.

- **Player.cpp:**

Define function for class Player, contain all the function that decide how to turn to the next move and which cell it go for including both human and AI.

Each object of class Player consist of **three attributes:** cards[] contain all available cards; num_card is number of available card; is_user will let you know who play

- **Games.cpp:**

Define function for class Game, contain all the functions for playing in games and make decision for the winner.

Each object of class Games consist of **three attributes:** board is object of class Board; user and ai are object of class Player.

- **Node.cpp:**

Define function for class Node, assign the Alpha-Beta Pruning search technique based Minimax Algorithm.

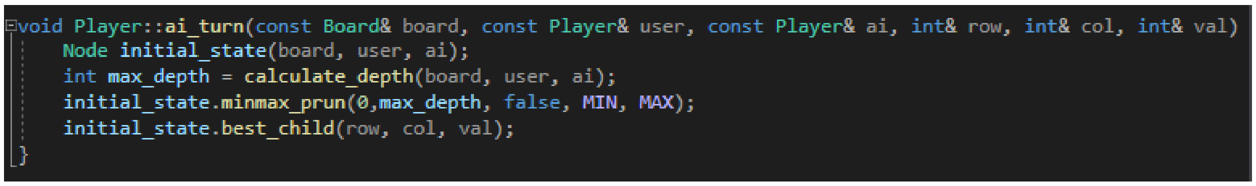
Each object of class Node include 1 board, user, ai, score, new_row and new_col and new_val to save the best move given by Alpha-Beta Pruning, children saves all branching factors of each node.

- **Header.h:**

Define all header file of 4 classes, store all methods of four above class.

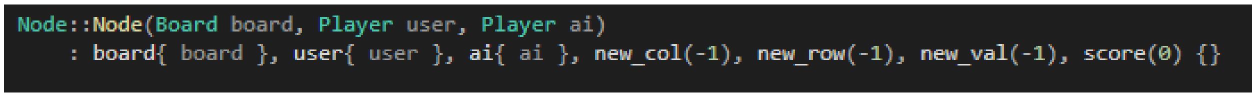
5. Some details about Search process

When AI turn, program will call function *ai.ai_turn(board, user, ai, row, col, val).*



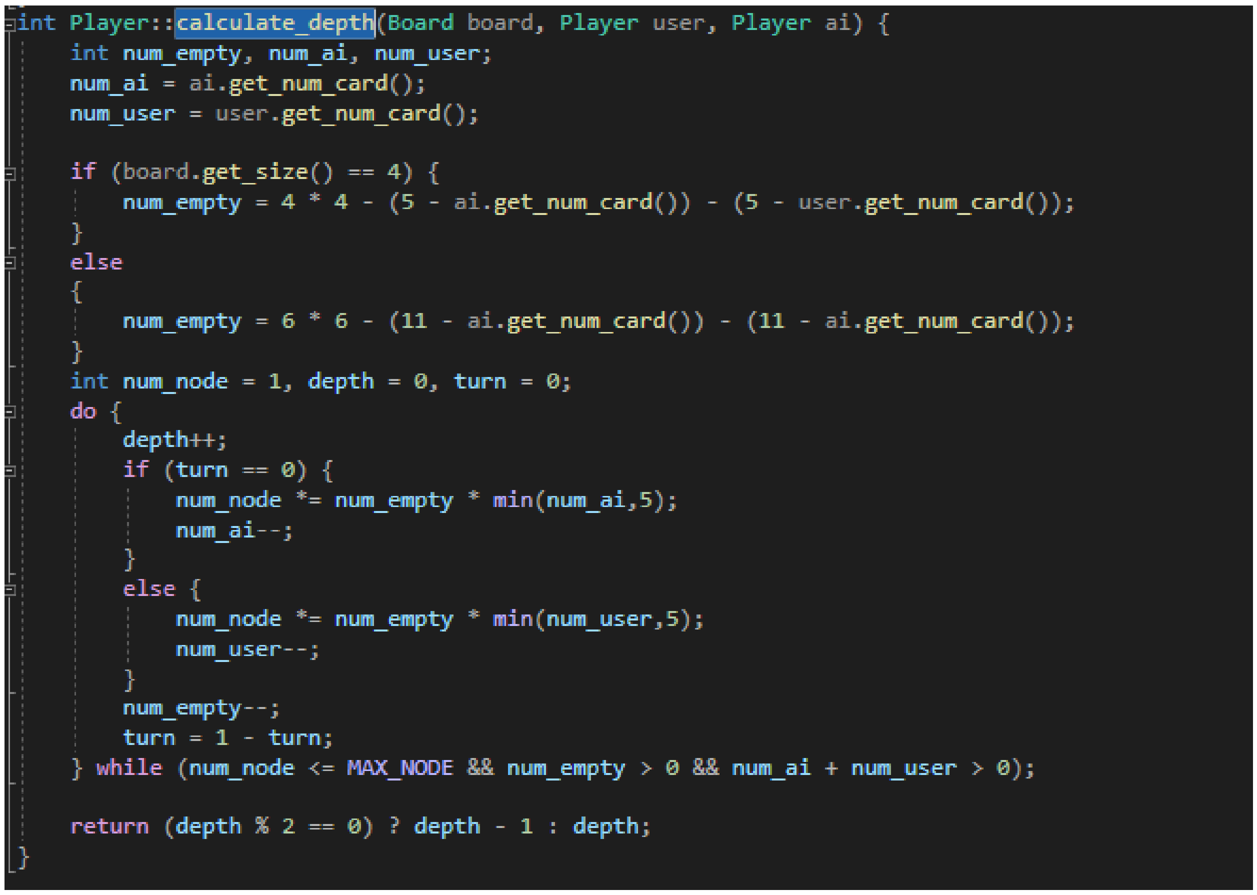
```
void Player::ai_turn(const Board& board, const Player& user, const Player& ai, int& row, int& col, int& val)
    Node initial_state(board, user, ai);
    int max_depth = calculate_depth(board, user, ai);
    initial_state.minmax_prun(0,max_depth, false, MIN, MAX);
    initial_state.best_child(row, col, val);
}
```

Firstly, it will create root Node of tree search by calling constructor of class Node.



```
Node::Node(Board board, Player user, Player ai)
    : board{ board }, user{ user }, ai{ ai }, new_col(-1), new_row(-1), new_val(-1), score(0) {}
```

Max_depth will be gotten from *calculate_depth(board, user,ai)* function.

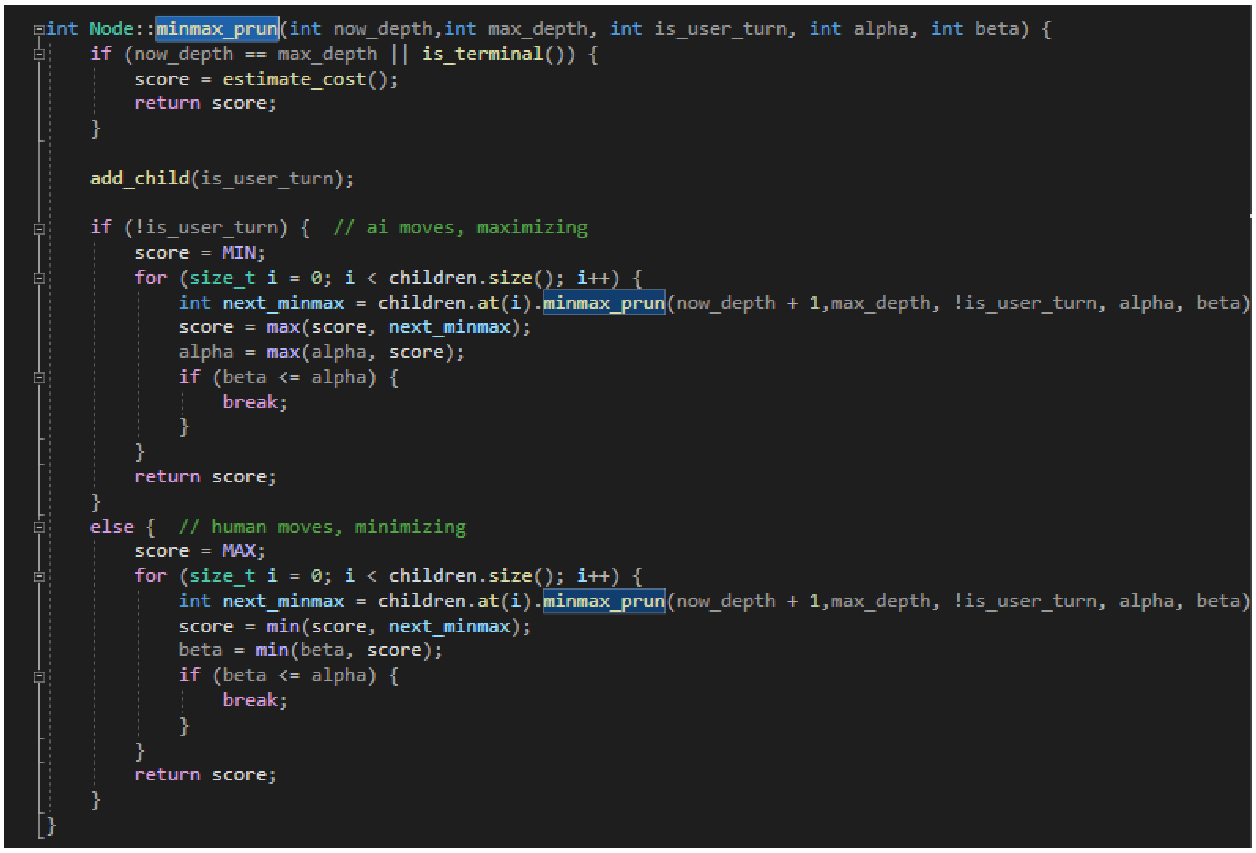


```
int Player::calculate_depth(Board board, Player user, Player ai) {
    int num_empty, num_ai, num_user;
    num_ai = ai.get_num_card();
    num_user = user.get_num_card();

    if (board.get_size() == 4) {
        num_empty = 4 * 4 - (5 - ai.get_num_card()) - (5 - user.get_num_card());
    }
    else
    {
        num_empty = 6 * 6 - (11 - ai.get_num_card()) - (11 - ai.get_num_card());
    }
    int num_node = 1, depth = 0, turn = 0;
    do {
        depth++;
        if (turn == 0) {
            num_node *= num_empty * min(num_ai,5);
            num_ai--;
        }
        else {
            num_node *= num_empty * min(num_user,5);
            num_user--;
        }
        num_empty--;
        turn = 1 - turn;
    } while (num_node <= MAX_NODE && num_empty > 0 && num_ai + num_user > 0);

    return (depth % 2 == 0) ? depth - 1 : depth;
}
```

Tree search will call mimax_prun() function recursively.

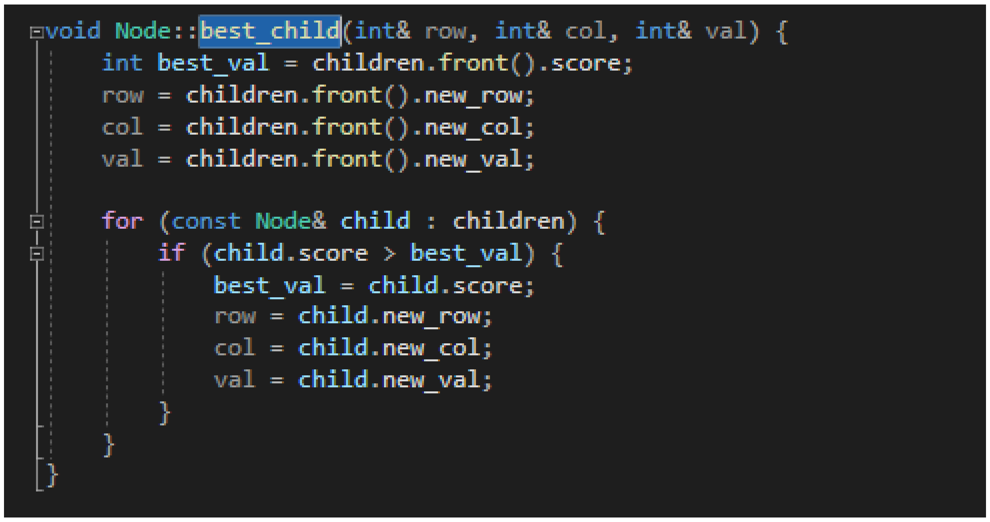


```
int Node::minmax_prun(int now_depth,int max_depth, int is_user_turn, int alpha, int beta) {
    if (now_depth == max_depth || is_terminal()) {
        score = estimate_cost();
        return score;
    }

    add_child(is_user_turn);

    if (!is_user_turn) {  // ai moves, maximizing
        score = MIN;
        for (size_t i = 0; i < children.size(); i++) {
            int next_minmax = children.at(i).minmax_prun(now_depth + 1,max_depth, !is_user_turn, alpha, beta)
            score = max(score, next_minmax);
            alpha = max(alpha, score);
            if (beta <= alpha) {
                break;
            }
        }
        return score;
    }
    else {  // human moves, minimizing
        score = MAX;
        for (size_t i = 0; i < children.size(); i++) {
            int next_minmax = children.at(i).minmax_prun(now_depth + 1,max_depth, !is_user_turn, alpha, beta)
            score = min(score, next_minmax);
            beta = min(beta, score);
            if (beta <= alpha) {
                break;
            }
        }
        return score;
    }
}
```

Get best move by function best_child()



```
void Node::best_child(int& row, int& col, int& val) {
    int best_val = children.front().score;
    row = children.front().new_row;
    col = children.front().new_col;
    val = children.front().new_val;

    for (const Node& child : children) {
        if (child.score > best_val) {
            best_val = child.score;
            row = child.new_row;
            col = child.new_col;
            val = child.new_val;
        }
    }
}
```

P/S: If you do have any problem with my report, don't hesitate to contact me via ***hungle0804@gmail.com***