Bhoos Smartbot 2023

TEAM: TopG [G for Geeks]

Team members:

1. Kushal Subedi [3rd year BCT, Pulchowk Campus]
2. Manoj Khatri [3rd year BCT, Pulchowk Campus]

Bid and Choose Trump:

<u>For bidding:</u>

1. Find out the highest repeated suit
2. Provide offset according to number of same suit as that of highest suit (1,2,3,4):

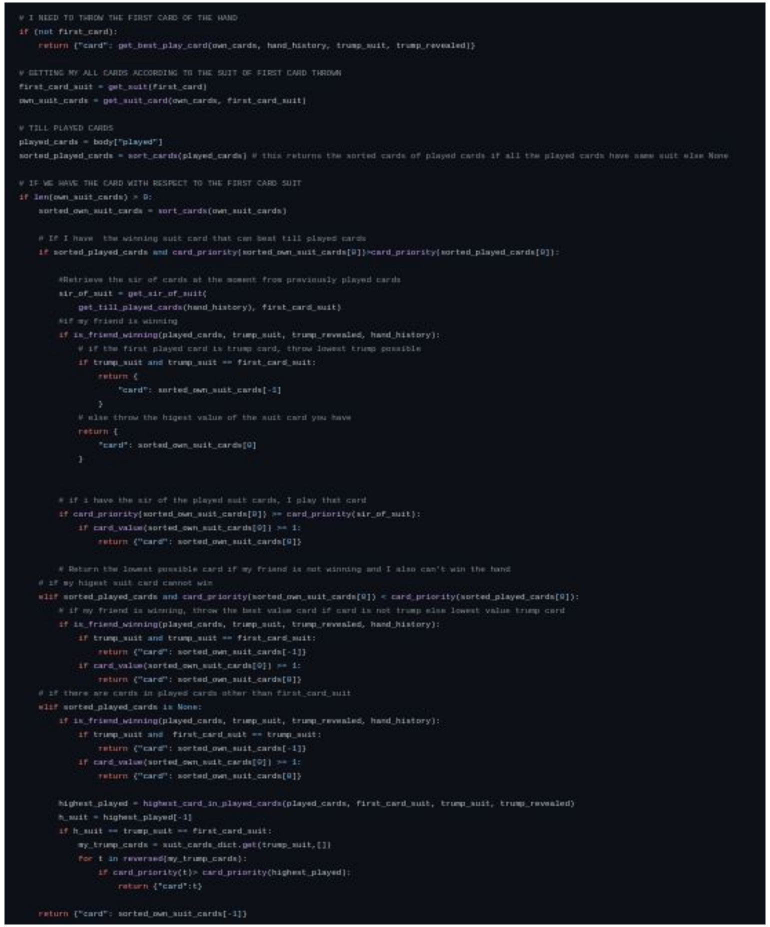
   For highest_count 4, 3, 2, 1, the offset is 17, 16, 15, 14 respectively
3. Bid = offset value + 50% of most repeated suit cards value + 34% of remaining cards value
4. Return rounded bid
5. The parameters are empirical

<u>For choosing trump:</u>

1. Choose suit with max number of repetition.
2. In case of multiple suits having save count, the suit with highest value in the cards

Technique 1: Rule based

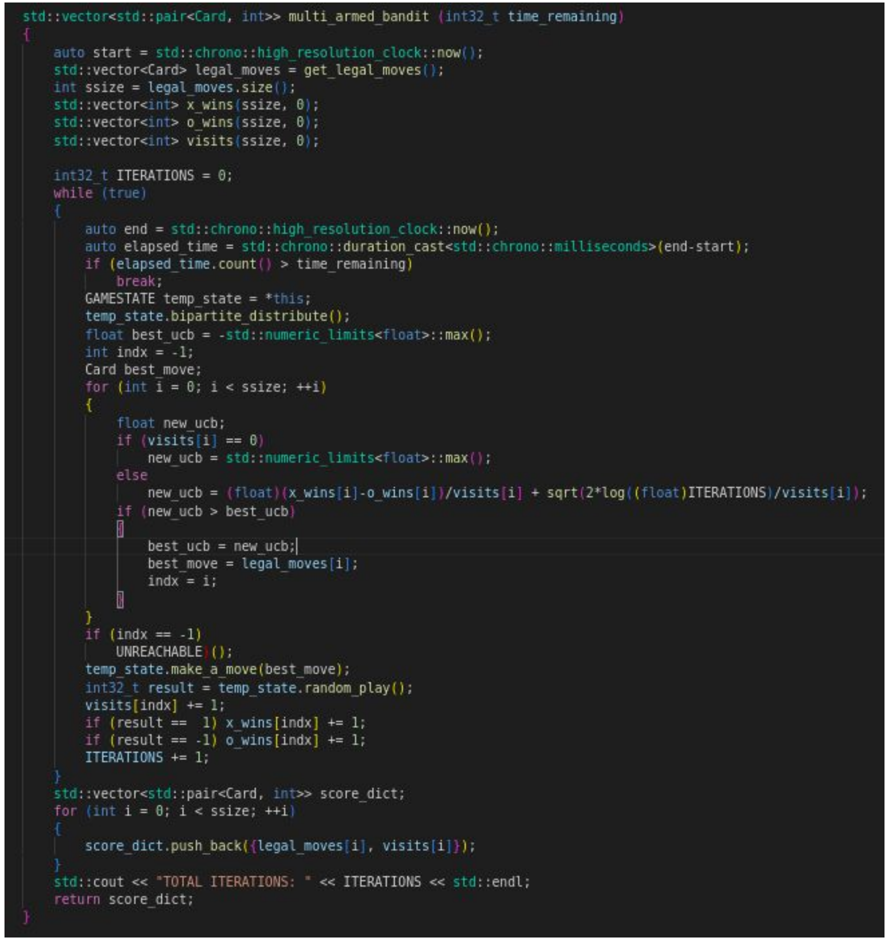
1. Play winning cards, if possible else lowest.
2. If friend is winning, throw the highest value card but in case of trump throw lowest trump.
3. Open trump only if value cards in currently played cards.
4. Never open trump if friend is winning.



```python
# I NEED TO THROW THE FIRST CARD OF THE HAND
if (not first_card):
    return {"card": get_best_play_card(own_cards, hand_history, trump_suit, trump_revealed)}

# GETTING MY ALL CARDS ACCORDING TO THE SUIT OF FIRST CARD THROWN
first_card_suit = get_suit(first_card)
own_suit_cards = get_suit_card(own_cards, first_card_suit)

# TILL PLAYED CARDS
played_cards = body["played"]
sorted_played_cards = sort_cards(played_cards) # this returns the sorted cards of played cards if all the played cards have same suit else None

# IF WE HAVE THE CARD WITH RESPECT TO THE FIRST CARD SUIT
if len(own_suit_cards) > 0:
    sorted_own_suit_cards = sort_cards(own_suit_cards)

    # If I have  the winning suit card that can beat till played cards
    if sorted_played_cards and card_priority(sorted_own_suit_cards[0])>card_priority(sorted_played_cards[0]):

        #Retrieve the sir of cards at the moment from previously played cards
        sir_of_suit = get_sir_of_suit(
            get_till_played_cards(hand_history), first_card_suit)
        #if my friend is winning
        if is_friend_winning(played_cards, trump_suit, trump_revealed, hand_history):
            # if the first played card is trump card, throw lowest trump possible
            if trump_suit and trump_suit == first_card_suit:
                return {
                    "card": sorted_own_suit_cards[-1]
                }
            # else throw the higest value of the suit card you have
            return {
                "card": sorted_own_suit_cards[0]
            }


        # if i have the sir of the played suit cards, I play that card
        if card_priority(sorted_own_suit_cards[0]) >= card_priority(sir_of_suit):
            if card_value(sorted_own_suit_cards[0]) >= 1:
                return {"card": sorted_own_suit_cards[0]}

        # Return the lowest possible card if my friend is not winning and I also can't win the hand
    # if my higest suit card cannot win
    elif sorted_played_cards and card_priority(sorted_own_suit_cards[0]) < card_priority(sorted_played_cards[0]):
        # if my friend is winning, throw the best value card if card is not trump else lowest value trump card
        if is_friend_winning(played_cards, trump_suit, trump_revealed, hand_history):
            if trump_suit and trump_suit == first_card_suit:
                return {"card": sorted_own_suit_cards[-1]}
            if card_value(sorted_own_suit_cards[0]) >= 1:
                return {"card": sorted_own_suit_cards[0]}
    # if there are cards in played cards other than first_card_suit
    elif sorted_played_cards is None:
        if is_friend_winning(played_cards, trump_suit, trump_revealed, hand_history):
            if trump_suit and  first_card_suit == trump_suit:
                return {"card": sorted_own_suit_cards[-1]}
            if card_value(sorted_own_suit_cards[0]) >= 1:
                return {"card": sorted_own_suit_cards[0]}

        highest_played = highest_card_in_played_cards(played_cards, first_card_suit, trump_suit, trump_revealed)
        h_suit = highest_played[-1]
        if h_suit == trump_suit == first_card_suit:
            my_trump_cards = suit_cards_dict.get(trump_suit,[])
            for t in reversed(my_trump_cards):
                if card_priority(t)> card_priority(highest_played):
                    return {"card":t}

    return {"card": sorted_own_suit_cards[-1]}
```

Technique 2: Multi-armed bandit

1. For each iterations:
2. Play the move for which bandit is maximum.
3. Randomly simulate.
4. Update the scores for each move.



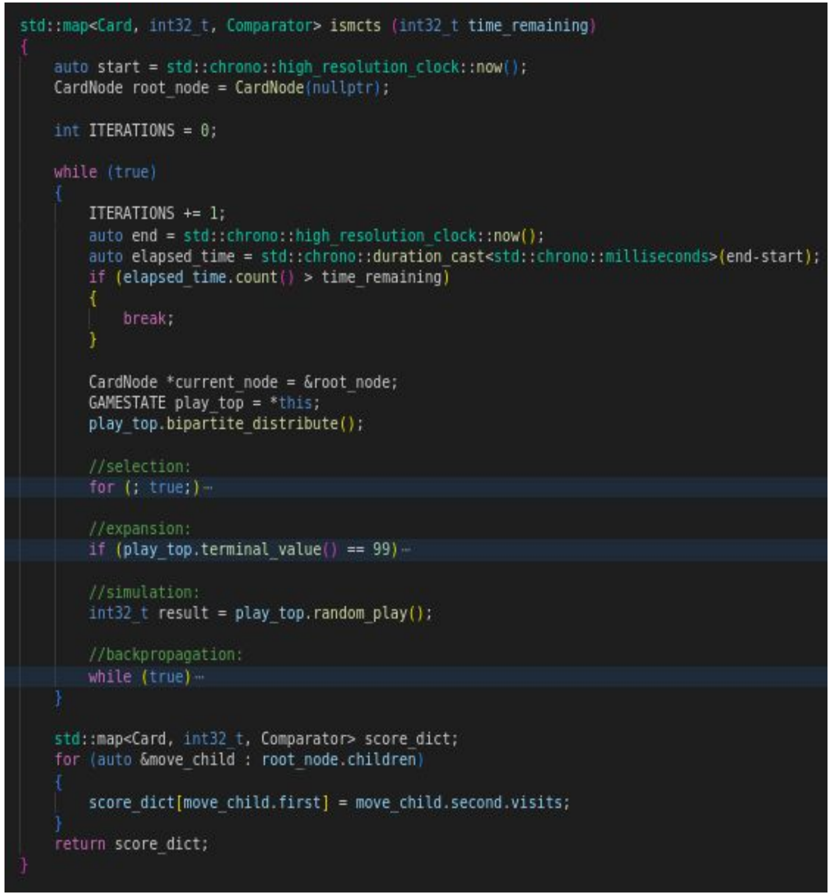
```cpp
std::vector<std::pair<Card, int>> multi_armed_bandit (int32_t time_remaining)
{
    auto start = std::chrono::high_resolution_clock::now();
    std::vector<Card> legal_moves = get_legal_moves();
    int ssize = legal_moves.size();
    std::vector<int> x_wins(ssize, 0);
    std::vector<int> o_wins(ssize, 0);
    std::vector<int> visits(ssize, 0);

    int32_t ITERATIONS = 0;
    while (true)
    {
        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
        if (elapsed_time.count() > time_remaining)
            break;
        GAMESTATE temp_state = *this;
        temp_state.bipartite_distribute();
        float best_ucb = -std::numeric_limits<float>::max();
        int indx = -1;
        Card best_move;
        for (int i = 0; i < ssize; ++i)
        {
            float new_ucb;
            if (visits[i] == 0)
                new_ucb = std::numeric_limits<float>::max();
            else
                new_ucb = (float)(x_wins[i]-o_wins[i])/visits[i] + sqrt(2*log((float)ITERATIONS)/visits[i]);
            if (new_ucb > best_ucb)
            {
                best_ucb = new_ucb;
                best_move = legal_moves[i];
                indx = i;
            }
        }
        if (indx == -1)
            UNREACHABLE();
        temp_state.make_a_move(best_move);
        int32_t result = temp_state.random_play();
        visits[indx] += 1;
        if (result ==  1) x_wins[indx] += 1;
        if (result == -1) o_wins[indx] += 1;
        ITERATIONS += 1;
    }
    std::vector<std::pair<Card, int>> score_dict;
    for (int i = 0; i < ssize; ++i)
    {
        score_dict.push_back({legal_moves[i], visits[i]});
    }
    std::cout << "TOTAL ITERATIONS: " << ITERATIONS << std::endl;
    return score_dict;
}
```

Technique 3: Determinized MCTS

1. Sample a determinization
2. For some n:
    a. Play the move for which, bandit is maximum
    b. Also create a node
    c. Simulate from the newly created node
    d. Update the scores towards to root node
3. Play the move with highest visits

Technique 4: ISMCTS

1. Sample a determinization
2. Play the move for which, bandit is maximum
3. Also create a node
4. Simulate from the newly created node
5. Update the scores towards to root node
6. Play the move with highest visits



```cpp
std::map<Card, int32_t, Comparator> ismcts (int32_t time_remaining)
{
    auto start = std::chrono::high_resolution_clock::now();
    CardNode root_node = CardNode(nullptr);

    int ITERATIONS = 0;

    while (true)
    {
        ITERATIONS += 1;
        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
        if (elapsed_time.count() > time_remaining)
        {
            break;
        }

        CardNode *current_node = &root_node;
        GAMESTATE play_top = *this;
        play_top.bipartite_distribute();

        //selection:
        for (; true;)⋯

        //expansion:
        if (play_top.terminal_value() == 99)⋯

        //simulation:
        int32_t result = play_top.random_play();

        //backpropagation:
        while (true)⋯
    }

    std::map<Card, int32_t, Comparator> score_dict;
    for (auto &move_child : root_node.children)
    {
        score_dict[move_child.first] = move_child.second.visits;
    }
    return score_dict;
}
```

Technique 5: MCTS + POM

1. Sample a determinization
2. Play the move for which, bandit is maximum
3. Also create a node (**only if player is you**)
4. Simulate from the newly created node
5. Update the scores towards to root node
6. Play the move with highest visits

Technique 6: MCTS + MOM

1. Sample a determinization
2. Play the move for which, bandit is maximum
3. Create a node (i**n separate trees for each)**
4. Simulate from the newly created node
5. Update the scores towards to root node
6. Play the move with highest visits

ISMCTS > MO-MCTS > Determinization MCTS >> POM-MCTS > Multi bandits >> Rule-based

Bandits:

UCB1:

$$A_t = \frac{\text{wins}}{\text{visits}} + \sqrt{\frac{2 * \ln(\text{availability})}{\text{visits}}}$$

Thomson sampling (Beta distribution)

$$A_t = \text{sample} \sim \text{Beta}(\text{wins}, \text{loses})$$

UCB tuned:

$$v_{\text{bound}} = \text{mean} - \text{mean}^2 + \sqrt{\frac{2 * \ln(\text{availability})}{\text{visits}}}$$

$$A_t = \text{mean} + \sqrt{\frac{v_{\text{bound}} * \ln(\text{availability})}{\text{visits}}}$$

<u>Thomson sampling > UCB tuned ~ UCB1 (c = 2)</u>

Optimizations:

1. Determinations:

- Bipartite matching of cards to each players according to their lost suits (DFS instead of BFS)
- Guess the missing suits from the play (not suit if sacrifices the valued card)

2. Simulation policy:

- Added heuristic for opponent play during simulations
- Trump guessing: assign probability to each suit based on cards played by bidder

3. Early stopping:

- No more visiting the branch, if gives loses for 90% of the time in 100 iterations

Optimizations (contd…)

3. Programming optimizations:

- **Python** for initial phase, **node.js** upto top8, and then **C++** for finals.
- Own hash functions for speed

THANK YOU BHOOS FOR THIS OPPORTUNITY.

TO OUR FELLOW PARTICIPANTS, IT WAS GREAT TO COMPETE WITH YOU.