Bhoos Smart Bots 2023

T-Rex

Sagar Kumar Thapa

076BCT, Pulchowk Campus

Diwakar Gyawali

076BCT, Pulchowk Campus

Our Journey

- Rule Based Approach
    - Python
- Determinized UCT - MCTS
    - Python
    - JavaScript
- Information Set Monte Carlo Tree Search
    - JavaScript
    - C++

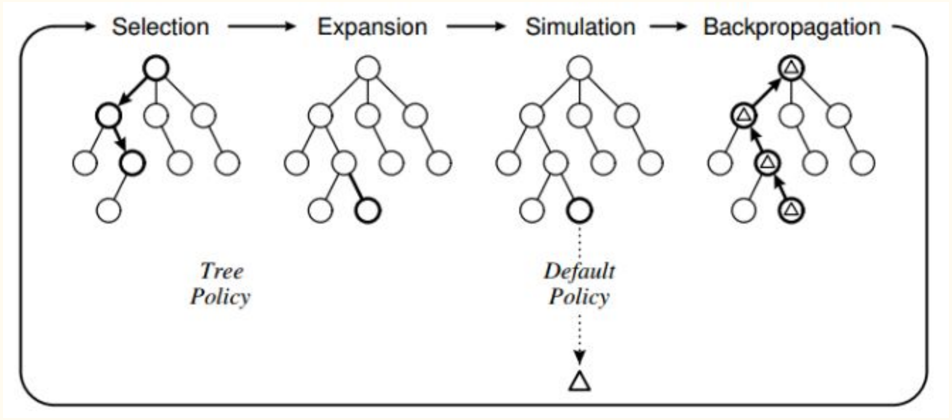
Rule Based Approach

- Bidding & Choosing Trump
    - Based on Max Suit Count
    - Presence of Value Cards (Jack, Nine, Ace, Ten)
- Game Play
    - If First Card
        - V1 : Play either Jack or any Zero Value Card
        - V2 : V1 & Inference of depleted suits
    - Else
        - Play Winning Valid Cards (Suit_To_Play Cards) with inference of depleted cards
        - Play Trump Cards if value cards are on current trick
        - Reveal Trump if partner is not winning

Programming Language Used - Python

MCTS (Monte Carlo Tree Search)

- Game tree search algorithm
- Use of simulated games to evaluate non-terminal states.
- Simulated games select random actions until a terminal state is reached and the reward is averaged over multiple simulations to estimate the strength of each action.



- 29 Point is an imperfect information game.

Approach 1 : Determinized - UCT

- Determinization
  - Distribute remaining cards to other players to create a state.
- Build a tree rooted at this state.
- Chooses a move for which the number of visits from the root, summed across all trees, is maximal.
- During Selection of Non-terminal nodes:
  - Choose the best child node using UCB1 algorithm.
  - UCB1 calculates the score of a node as
    - $(avg\_Score) + c*\sqrt{(\log n/n_i)}$

      where, n = number of visits in parent node, n_i = number of visits in the child node

      C = exploration constant

Programming Languages Used : Python & Js

Approach 2 : Information Set MCTS

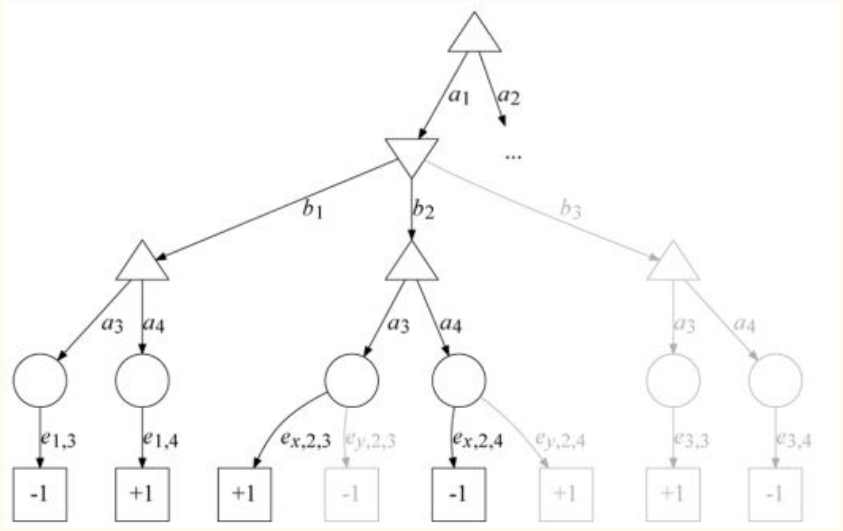
- Determinization
  - Distribute remaining cards to other players to create a state.
- Build and maintain a single tree.
- In a single determination, increase a single node in the tree.

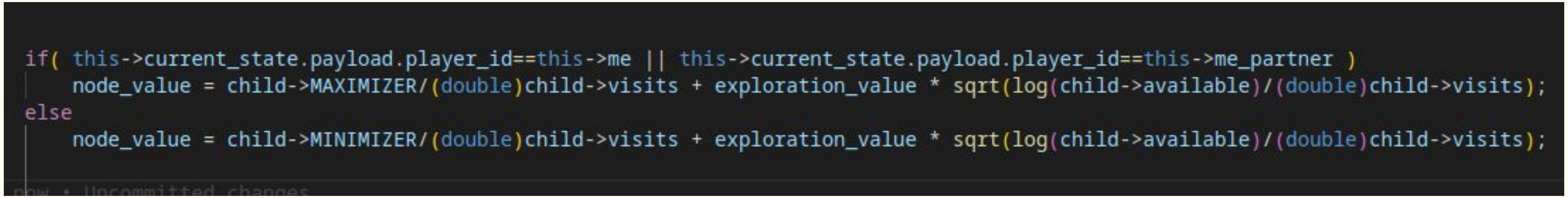
- During Selection of Non-terminal nodes:
  - Choose the best child node using UCB1 algorithm.
  - UCB1 calculates the score of a node as
    - $(avg\_Score) + c*\sqrt{(\log n/n_i)}$

      where, n = number of visits in parent node when the child is available, n_i = number of visits in the child node

      C = exploration constant

Programming Languages Used : Python & C++



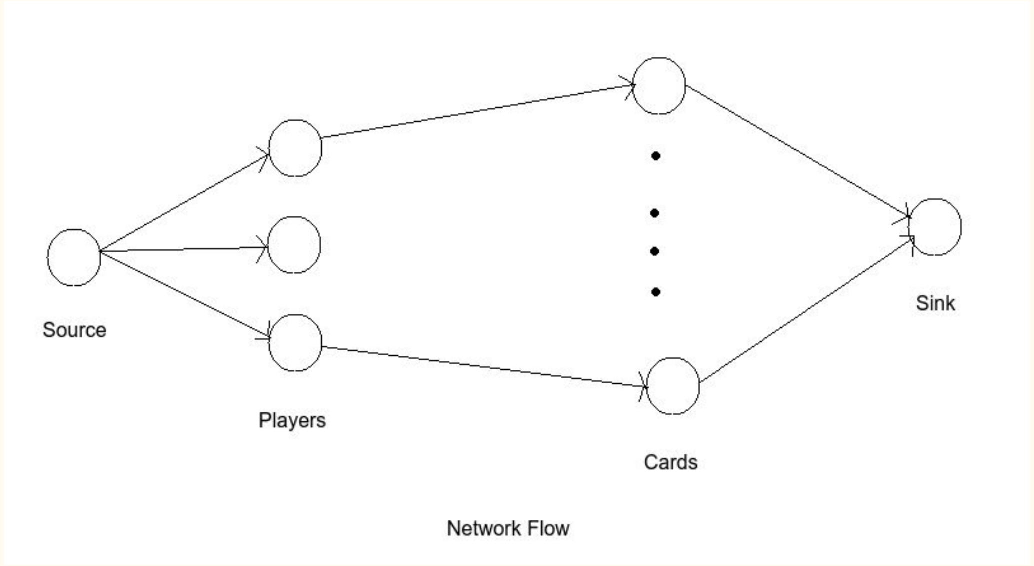


```
if( this->current_state.payload.player_id==this->me || this->current_state.payload.player_id==this->me_partner )
    node_value = child->MAXIMIZER/(double)child->visits + exploration_value * sqrt(log(child->available)/(double)child->visits);
else
    node_value = child->MINIMIZER/(double)child->visits + exploration_value * sqrt(log(child->available)/(double)child->visits);
```

Optimization : Determinization

- V1 : Inference of Depleted Suits for each player & distribute if possible else distribute randomly.
- V2:
  - Designed a Bipartite Graph
  - Converted to a Network Flow
  - Maximized Flow
  - Distributed Cards

BFS technique to find the augmenting path

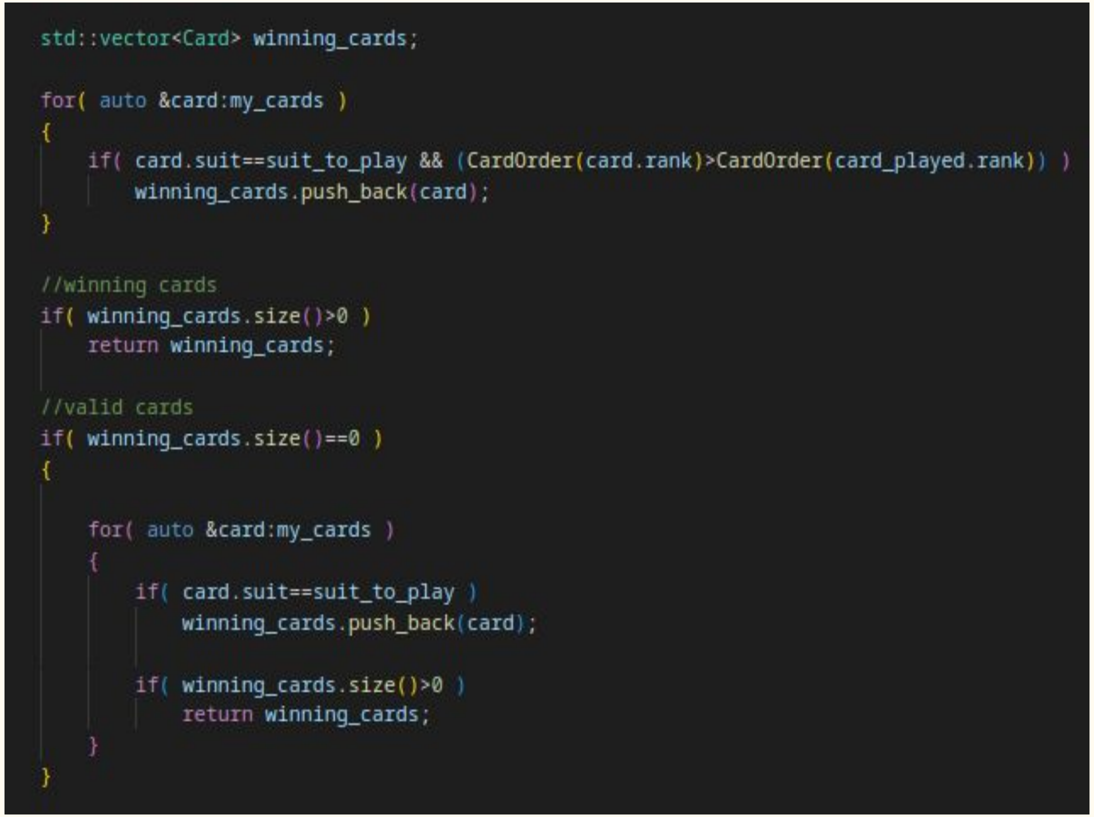


Optimization : Rollout Policy

- Modified Heuristic used in Rule Based Approach
    - Grouping Winning Cards
    - Saving Trumps if Bidder

Early Return if win/loss is sure to pump iterations.

Reward Scheme:
- 1 for win
- -1 for loss
- 0 for nullified


```
std::vector<Card> winning_cards;

for( auto &card:my_cards )
{
    if( card.suit==suit_to_play && (CardOrder(card.rank)>CardOrder(card_played.rank)) )
        winning_cards.push_back(card);
}

//winning cards
if( winning_cards.size()>0 )
    return winning_cards;

//valid cards
if( winning_cards.size()==0 )
{

    for( auto &card:my_cards )
    {
        if( card.suit==suit_to_play )
            winning_cards.push_back(card);

        if( winning_cards.size()>0 )
            return winning_cards;
    }
}
```

Unsuccessful Attempts

- Rule Based Approach

- Determinized-UCT

- Guessing trump suit for each determinization based on the sample space of the cards to be distributed.

- Tree Pruning
    - Saving trump cards & obviously playable cards.

- Single Observer Information Set Monte Carlo Tree Search + Partially Observable Moves
    - The implementation was weak because of weak opponent modeling.

We shifted from Python to JavaScript and from JavaScript to C++ to pump up the iterations.

Thank You