WAR: WAR with Auction Rounds

Report for *Game Theory & Networks* with *Dr. Dejun Yang*

Jack Rosenthal
`jrosenth@mines.edu`

Paul Sattizahn
`psattiza@mines.edu`

***Abstract*—War is a probabilistic card game commonly played in North America. In this paper, we investigate our own strategic variant of War called WAR (which stands for WAR with Auction Rounds) from a game theoretic approach, design strategies for playing the game, and analyze the effectiveness of the strategies using a computer simulation.**

I. Introduction

War is a card game played with two players and a single deck of cards. The deck of cards is shuffled, and each player is dealt half of the deck. The players repeatedly flip the top card of their deck, the player flipping the higher valued card wins both cards and puts the cards in their *wins pile*. If the two players play the same card, they "go to war". In a war (the resolution of a tie), both players discard the top three cards from their deck to the pot, then play a fourth. The fourth card decides who wins the war. If the fourth card is the same, the players go to war again until the war is resolved. Once a player runs out of cards in their deck, they flip their wins pile and continue to play. If they have no wins pile, the other player wins.

Studying War would be very boring from a game theoretical approach, as the game involves no element of strategy. Instead, we propose a modified game: WAR – a recursive acronym standing for **WAR with Auction Rounds**. To distinguish these two games, we will refer to the original game of War as *Regular War*, and our modified version as *WAR* (as printed in all capital letters).

WAR is a game played with N players, each player starts with one suite of cards (12 cards, labelled $2, 3, \ldots, 10, \mathrm{J}, \mathrm{Q}, \mathrm{K}$). The cards J, Q, and K have the values 11, 12, and 13, respectively. The players then repeatedly *choose* a card from their hand to play, the cards are then flipped simultaneously and the player who plays the highest card takes all the cards for their wins pile. Similar to Regular War, when there is a tie for the highest card, all the players who played the highest card go to war. Unlike Regular War, during a war, each of the players involved get to *choose* which three cards to discard and which card to battle with. This process is repeated until the war is resolved. Finally, just like Regular War, when a player is out of cards in their hand, they pick up their wins pile if they have one, otherwise, they are out of the game.

Normally, WAR is won by the last remaining player in the game. However, there does exist the possibility that a game of WAR cannot be won by any of the players. This will occur when all of the remaining players get in a tie, but all players run out of cards before the tie is resolved.

To assist in your understanding of the game, the authors have produced a video demonstration of the game. This video is available online at:

https://www.youtube.com/watch?v=KzWVCbNsjwc

In this paper, we will define a model for the game, and discuss various strategies (and their relation in terms of effectiveness to the other strategies) we discovered.

II. System Model

WAR is a repeated static game; that is, all players move simultaneously, the utility of one player depends on the decision of another player, and the state of the game is not independent from round to round.

At the beginning of a game of WAR, every player begins with a known set of cards, and after all of the events that occur in the game, the cards that are exchanged are known to every other player. In theory, if every player were able to count the cards that went by, they could determine what any other player has in their hand and in their wins pile. We say that WAR is a game of *perfect information*, all players have complete knowledge of the state of the game (with the exception of the information of the card another player is about to play, which means our game differs slightly from the standard definition of perfect information).

We will use the following mathematical notation in our strategies to describe the system:

1) N: The number of players in the game, where each player is identified with a number n, where $n \in \{1, \ldots, N\}$.
2) H_n: The set of cards in player n's hand.
3) W_n: The set of cards in player n's wins pile.

In addition, we will define a utility function $u_n(H_n, W_n)$ which can not only be used to evaluate the utility of a player at the beginning of a game, but also evaluate how well a player is doing during the middle of a game.

$$u_n(H_n, W_n) = \sum_{i=1}^{|H_n|} H_{ni} + \sum_{j=1}^{|W_n|} W_{nj}$$

It follows that $u_n(H_n, W_n)$ for the winning player at the end of the game is $N \sum_{i=2}^{13} i = 90N$ and the utility for all other players is 0. In the case when nobody wins, the utility for all players is 0.

Figure 1 formally defines a game of war using pseudocode.

III. Strategies

To further analyze the game, we define strategies. Each strategy is assigned to a type of player, such that all players of a given type will play WAR with the corresponding strategy.

A. MinPlayer

The MinPlayer strategy will always pick its minimum card when given a choice. In other words, for a MinPlayer n, SelectCard(n, H_n) = $\min(H_n)$. In the case of a war (tie), the MinPlayer will discard its lowest three cards, and then it selects its next lowest card to play. In other words, for a MinPlayer n, SelectTie(n, H_n) = ($\min_1(H_n)$, $\min_2(H_n)$, $\min_3(H_n)$, $\min_4(H_n)$).

B. MaxPlayer

The MaxPlayer strategy will always pick its maximum card when given a choice. In other words, for a MinPlayer n, SelectCard(n, H_n) = $\max(H_n)$. In the case of a war (tie), the MaxPlayer strategy will discard similarly to a MinPlayer. The MaxPlayer discards its lowest three cards, but it selects its highest card to play. In other words, for a MaxPlayer n, SelectTie(n, H_n) = ($\min_1(H_n)$, $\min_2(H_n)$, $\min_3(H_n)$, $\max(H_n)$).

C. DummiePlayer

The DummiePlayer implements a very simple strategy. Whenever the DummiePlayer has the option between a set of choices, it will choose one of the choices randomly, where each choice has an equal chance being picked. In other words, for a DummiePlayer n, SelectCard(n, H_n) = RandomChoice(H_n). The

```
procedure WAR(N)
    ▷ Initialize the hand and wins pile of all players
    for n in {1, ..., N} do
        H_n ← {2, ..., 13}
        W_n ← ∅
    end for
    z ← ∅
    ▷ While there are still remaining players...
    while | {H_n : n ∈ {1, ..., N} | |H_n| ⩾ 1} | ⩾ 2 do
        p ← [n : n ∈ {1, ..., N} | |H_n| ⩾ 1]
        ▷ Ask each player to select a card
        c ← []
        for all n in p do
            c_n ← SelectCard(n, H_n)
        end for
        for all n in p do
            Remove c_n from H_n and reveal
        end for
        w ← [c_n | c_n ∈ max(c)]
        z ← z ∪ c
        if |w| ≠ 1 then
            ▷ It's a tie. Go to war!
            while |w| ⩽ 2 do
                c ← []
                for all n in w do
                    if |H_n| ⩽ 3 then
                        z ← z ∪ H_n
                        H_n ← ∅
                        Remove n from w and continue
                    end if
                    d_1, d_2, d_3, c_n ← SelectTie(n, H_n)
                    z ← z ∪ {d_1, d_2, d_3, c_n}
                end for
                for all n in c do
                    Remove c_n from H_n and reveal
                end for
                w ← [c_n | c_n ∈ max(c)]
            end while
        end if
        if |w| = 1 then
            W_w1 ← W_w1 ∪ z
            z ← ∅
        end if
        for all n in p do
            if H_n = ∅ then
                H_n ← W_n
                W_n ← ∅
            end if
        end for
    end while
end procedure
```

Fig. 1. A game of WAR, represented formally in pseudocode.

cards to discard in the case of a tie are also picked randomly. In other words, for a DummiePlayer n, $\textsc{SelectTie}(n,\, H_n) = \textsc{RandomSample}(4,\, H_n)$.

D. SimpleMindedPlayer1

The SimpleMindedPlayer1 is a computationally simple strategy that preforms well against human players unaware of the strategy, as well as against DummiePlayers. The strategy is defined as follows:

Determination of a normal play (that is, $\textsc{SelectCard}(n,\, H_n)$) is defined by the following algorithm:

1) Compute $S = \{H_{ic} \,|\, \forall j, d\, [H_{ic} > H_{jd}]\}$.
2) If $S \neq \emptyset$, play $\min(S)$.
3) Otherwise if, $|H_i| \geqslant 5$, $\exists j, c\, [H_{ic} = \max(H_j) \wedge \max_2(H_i) \geqslant \max_2(H_j)]$, then play the corresponding H_{ic}.
4) Otherwise, play $\min(H_i)$.

To play a tie ($\textsc{SelectTie}(n,\, H_n)$), follow strategy similar to the algorithm described above, but remove 3 smallest cards from hand first to be discarded. In other words, for a SimpleMindedPlayer1 n,

$$D_n = \left\{ \min_1(H_n),\, \min_2(H_n),\, \min_3(H_n) \right\}$$
$$\textsc{SelectTie}(n,\, H_n) = (D_n,\, \textsc{SelectCard}(n, H_n - D_n))$$

To discover the effectiveness of the SimpleMindedPlayer1 strategy, we wrote a computer simulation. This simulation and the outcomes are presented in Section IV.

E. SimpleMindedPlayer2

The SimpleMindedPlayer2 strategy is similar to the SimpleMindedPlayer1, except it plays *more* risky when deciding weather to enter a potential tie. This strategy is defined as follows:

Determination of a normal play (that is, $\textsc{SelectCard}(n,\, H_n)$) is defined by the following algorithm:

1) Compute $S = \{H_{ic} \,|\, \forall j, d\, [H_{ic} > H_{jd}]\}$.
2) If $S \neq \emptyset$, play $\min(S)$.
3) Otherwise if, $|H_i| \geqslant 5$, then play $\max(H_i - S)$.
4) Otherwise, play $\min(H_i)$.

To play a tie ($\textsc{SelectTie}(n,\, H_n)$), follow strategy similar to the algorithm described above, but remove 3 smallest cards from hand first to be discarded. In other words, for a SimpleMindedPlayer1 n,

$$D_n = \left\{ \min_1(H_n),\, \min_2(H_n),\, \min_3(H_n) \right\}$$
$$\textsc{SelectTie}(n,\, H_n) = (D_n,\, \textsc{SelectCard}(n, H_n - D_n))$$

To discover the effectiveness of the SimpleMindedPlayer2 strategy, we wrote a computer simulation. This simulation and the outcomes are presented in Section IV.

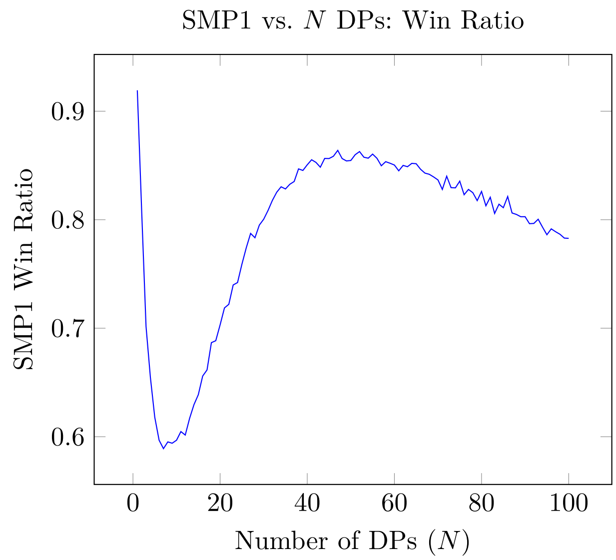

Fig. 2. Win Ratio for SimpleMindedPlayer1

F. GeneralPurposeAdversary

When the strategies of all other players in the game are *known*, a general purpose adversarial player can be designed to play against these known strategies. We call this strategy (appropriately) the GeneralPurposeAdversary.

In prose, this player computes the moves of every other player, and chooses the best card that can win the round (if it has one), or it's minimum card otherwise. In the case the GeneralPurposeAdversary could force a tie, the GeneralPurposeAdversary will recursively simulate the rounds of ties, and only force the round of tie(s) if it can win.

To select during a tie, the three lowest cards are discarded, and the selected card is computed using the function described above in prose.

A formal definition of this player can be found in our code submission (attached to the back of this report).

For any known player strategy and any amount of players, a GeneralPurposeAdversary player will always be able to at least tie, if not win, against the pool of players.

IV. Simulation

To investigate the effectiveness of the SimpleMindedPlayer1 and SimpleMindedPlayer2 strategies, we wrote a computer simulation of WAR. For each of the players, we simulated games from 1 to 100 DummiePlayer opponents, running over 9,000 (equal to 9,030) games per count of DummiePlayer. This totals to 1.8 million games simulated, totaling to about 1 month of computational time (distributed across 48 processor cores).

The results of this simulation are shown in Figures 2 and 5. Notice that SimpleMindedPlayer1 preforms

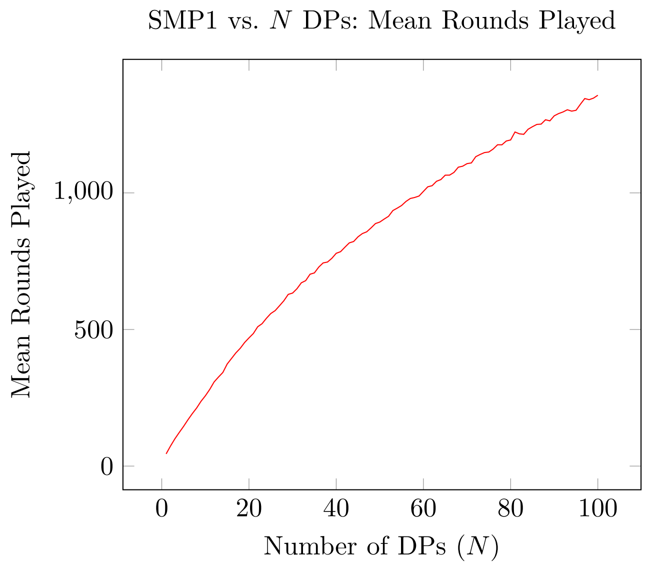

Fig. 3. Mean rounds played before victory for the SimpleMindedPlayer1

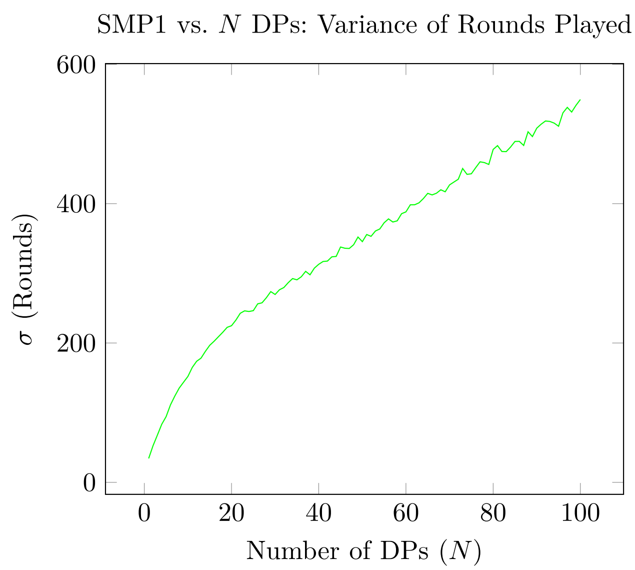

Fig. 4. Standard deviation of the rounds played before victory for the SimpleMindedPlayer1

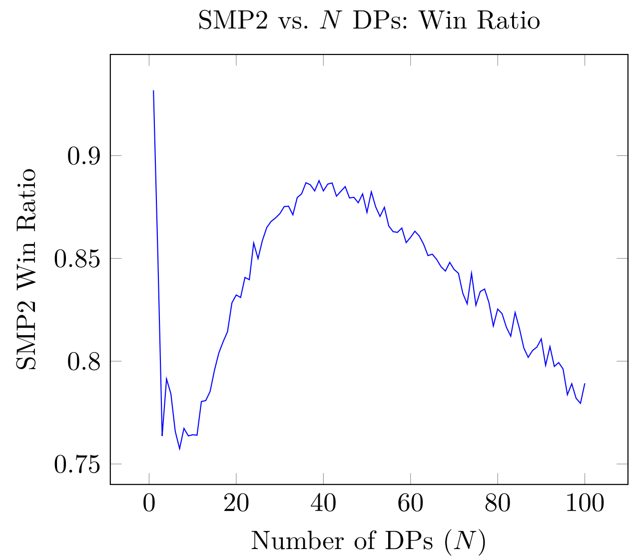

Fig. 5. Win Ratio for SimpleMindedPlayer2

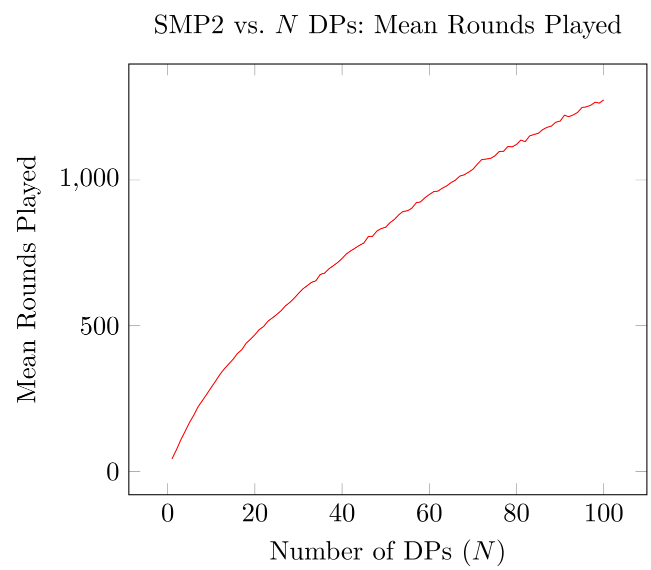

Fig. 6. Mean rounds played before victory for the SimpleMindedPlayer2

worse against a pool of DummiePlayers even though it takes less risk in its strategy. The more risky player, the SimpleMindedPlayer2, was able to capitalize on the imperfect play of the DummiePlayer, but may not preform as well with a more worthy opponent.

The mean number of rounds from our simulation to defeat N DummiePlayers are shown in Figures 3 and 6. The standard deviation is shown in Figures 4 and 7. We noticed no significant difference in the number of rounds played between the two strategies, even though the win ratio differed significantly.

V. Conclusion

WAR is an adaptation from a simple card game, Regular War (explained in the introduction). From our work, we were able to find three effective strategies, two of which are useful even when the strategies of the opponents are not known.

Our work falls short in our simulation methods: we tested against a DummiePlayer, however this is impractical in a real game of WAR, as the opponents will have (at least some) non-random strategy, and this strategy generally cannot be predicted. Given the resources, we would test our SimpleMindedPlayer1 and Sim-

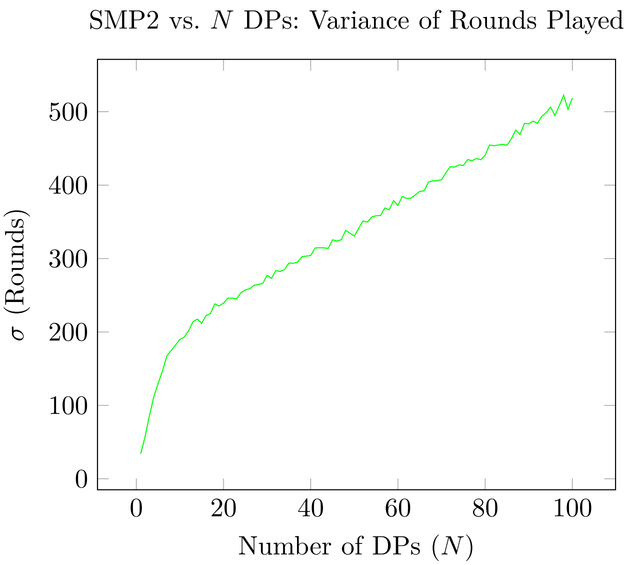

Fig. 7. Standard deviation of the rounds played before victory for the SimpleMindedPlayer2

pleMindedPlayer2 strategies in a real game against humans; however, considering the length of a game and the resources required to run a human simulation, testing this would be intractable.

From an academic/learning perspective, we are satisfied with the outcome of our work. We learned much in the process of writing the computer simulation and gathering data, as well as from the design of the strategies.

Appendix

Attached is our simulation code, referenced in the report. Enjoy!

```
import heapq
from random import Random
from itertools import count
from copy import deepcopy

class Player:
    amount_of_me = 0
    def __init__(self, name=None):
        if not name:
            self.__class__.amount_of_me += 1
            self.name = "{}{}".format(
                self.__class__.__name__,
                self.__class__.amount_of_me)
        else:
            self.name = name
        self.hand = list(range(2,14))
        self.wins = []

class DummiePlayer(Player):
    def __init__(self, *args, rngchoose=None, rngtie=None, **kwargs):
        super().__init__(*args, **kwargs)
        if not rngchoose:
            rngchoose = Random()
        if not rngtie:
            rngtie = Random()
        self.rngchoose = rngchoose
        self.rngtie = rngtie

    def chooseone(self, players):
        return self.rngchoose.randrange(len(self.hand))

    def choosetie(self, players):
        selection = self.rngtie.sample(range(len(self.hand)), 4)
        return selection[:-1], selection[-1]

class SimpleMindedPlayer(Player):
    def chooseone(self, players):
        self.hand.sort()
        max_of_all = max(max(p.hand) for p in players)
        max_of_me = self.hand[-1]
        if max_of_me > max_of_all:
            for i, card in enumerate(self.hand):
                if card > max_of_all:
                    return i
        if max_of_me == max_of_all and len(self.hand) >= 5:
            try:
                lookahead_max = max(
                    heapq.nlargest(2, p.hand)[1]
                    for p in players if len(p.hand) > 1)
            except ValueError:
                return len(self.hand) - 1
            my_next_best = heapq.nlargest(2, self.hand)[1]
            if my_next_best >= lookahead_max:
```

```
                return len(self.hand) - 1
        return 0

    def choosetie(self, players):
        if not any(p.hand for p in players):
            # we win this one no matter what happens
            return [0, 1, 2], 3
        upper_hand = self.hand[3:]
        max_of_all = max(max(p.hand) for p in players if p.hand)
        max_of_me = self.hand[-1]
        if max_of_me > max_of_all:
            for i, card in enumerate(upper_hand):
                if card > max_of_all:
                    break
            return [0, 1, 2], 3 + i
        if max_of_me == max_of_all and len(upper_hand) >= 5:
            try:
                lookahead_max = max(
                    heapq.nlargest(2, p.hand)[1]
                    for p in players if len(p.hand) > 1)
            except ValueError:
                return [0, 1, 2], len(self.hand) - 1
            my_next_best = heapq.nlargest(2, upper_hand)[1]
            if my_next_best >= lookahead_max:
                return [0, 1, 2], len(self.hand) - 1
        return [0, 1, 2], 3

class SimpleMindedPlayer2(Player):
    def chooseone(self, players):
        self.hand.sort()
        max_of_all = max(max(p.hand) for p in players)
        max_of_me = self.hand[-1]
        if max_of_me > max_of_all:
            for i, card in enumerate(self.hand):
                if card > max_of_all:
                    return i
        if max_of_me == max_of_all and len(self.hand) >= 5:
            return len(self.hand) - 1
        return 0

    def choosetie(self, players):
        if not any(p.hand for p in players):
            # we win this one no matter what happens
            return [0, 1, 2], 3
        upper_hand = self.hand[3:]
        max_of_all = max(max(p.hand) for p in players if p.hand)
        max_of_me = self.hand[-1]
        if max_of_me > max_of_all:
            for i, card in enumerate(upper_hand):
                if card > max_of_all:
                    break
            return [0, 1, 2], 3 + i
        if max_of_me == max_of_all and len(upper_hand) >= 5:
            try:
                lookahead_max = max(
                    heapq.nlargest(2, p.hand)[1]
```

```python
                    for p in players if len(p.hand) > 1)
            except ValueError:
                return [0, 1, 2], len(self.hand) - 1
            my_next_best = heapq.nlargest(2, upper_hand)[1]
            if my_next_best >= lookahead_max:
                return [0, 1, 2], len(self.hand) - 1
        return [0, 1, 2], 3

class GeneralPurposeAdversary(Player):
    def chooseone(self, players):
        self.hand.sort()
        max_of_all = 0
        max_players = []
        for p in players:
            if not p.hand:
                continue
            card = p.hand[p.chooseone([pp for pp in players if pp != p] + [self])]
            if card > max_of_all:
                max_of_all = card
                max_players = [p]
            elif card == max_of_all:
                max_players.append(p)
        max_of_me = self.hand[-1]
        if max_of_me > max_of_all:
            for i, card in enumerate(self.hand):
                if card > max_of_all:
                    return i
        if max_of_me == max_of_all and len(self.hand) >= 5:
            max_players = deepcopy(max_players)
            me = deepcopy(self)
            for p in max_players:
                p.hand.remove(max_of_all)
                if not p.hand:
                    max_players.remove(p)
            me.hand.remove(max_of_all)
            my_discards, my_choice = me.choosetie(max_players)
            max_choice = 0
            for p in max_players:
                if len(p.hand) < 4:
                    continue
                discard_idxs, choice_idx = p.choosetie([pp for pp in max_players if pp != p] + [me])
                if p.hand[choice_idx] > me.hand[my_choice]:
                    return 0
            return len(self.hand) - 1
        return 0

    def choosetie(self, players):
        if not any(p.hand for p in players):
            # we win this one no matter what happens
            return [0, 1, 2], 3
        me = deepcopy(self)
        me.hand = me.hand[3:]
        return [0, 1, 2], me.chooseone(players) + 3

class HumanPlayer(Player):
```

```
    def chooseone(self, players):
        self.hand.sort()
        print("==> {} <==".format(self.name))
        while True:
            print()
            if self.wins:
                print("Here is your wins pile (you cannot play now):")
                print(" " * 3, ', '.join(map(str, self.wins)))
                print()
            print("Here is your hand:")
            print(" " * 3, ', '.join(map(str, self.hand)))
            print()
            try:
                choice = int(input("Which card do you choose? "))
                idx = self.hand.index(choice)
                return idx
            except ValueError:
                print("That's not a valid choice, dummie!")

    def choosetie(self, players):
        print("==> {} <==".format(self.name))
        h = self.hand[:]
        idxs = []
        print()
        print("There was a tie between you and {} other player{}.".format(
                len(players), "s" if players != 1 else ""
            )
        )
        for prompt in ("Choose the first card to discard: ",
                       "Choose the second card to discard: ",
                       "Choose the third card to discard: ",
                       "Choose your play: "):
            print()
            print("Here is your hand:")
            print(" " * 3, ', '.join(
                ("\x1B[34m{}\x1B[0m" if th is not None else "\x1B[33m{}\x1B[0m").format(rh)
                for th, rh in zip(h, self.hand))
            )
            print()
            while True:
                try:
                    choice = int(input(prompt))
                    idx = h.index(choice)
                    break
                except ValueError:
                    print("That's not a valid choice, dummie!")
            h[idx] = None
            idxs.append(idx)
        return idxs[:-1], idxs[-1]

class InvalidMoveError(Exception):
    pass

def play_game(players, logger=print, kill_at_uniq=False):
    carry_pot = []
    for rnd in count(0):
```

```
        players_in_round = [p for p in players if p.hand]
        logger("Start of Round {}. Remaining players: {}.".format(
            rnd, ', '.join(p.name for p in players_in_round)
        ))
        if kill_at_uniq:
            for p in players_in_round:
                if type(p) != type(players_in_round[-1]):
                    break
            else:
                if not players_in_round:
                    return rnd, None
                return rnd, players_in_round[0]
        if len(players_in_round) < 2:
            logger("Less than two players remain. End of game.")
            break

        # Ask each player for their card of choice
        choices = []
        for p in players_in_round:
            idx = p.chooseone([op for op in players_in_round if op != p])
            if idx not in range(len(p.hand)):
                raise InvalidMoveError
            choices.append((p, idx))

        # Next, remove the cards from each hand and build the pot
        # Max card will be m
        pot = []
        m = 0
        for p, idx in choices:
            card = p.hand.pop(idx)
            logger("{} selects the {} at index {}.".format(p.name, card, idx))
            if card > m:
                m = card
            pot.append((p, card))
            carry_pot.append(card)

        winners = [p for p, card in pot if card == m]
        logger("Winners for this round are {}.".format(
            ', '.join(w.name for w in winners)
        ))
        while len(winners) > 1:
            logger("It's a tie!")
            choices = []

            for p in winners:
                if len(p.hand) < 4:
                    choices.append((p, list(range(len(p.hand))), None))
                    continue
                discard_idxs, choice_idx = p.choosetie([w for w in winners if w != p])
                if (any(idx not in range(len(p.hand))
                        for idx in discard_idxs + [choice_idx])
                    or len(set(discard_idxs + [choice_idx])) != 4):
                    raise InvalidMoveError
                choices.append((p, discard_idxs, choice_idx))

            pot = []
```

```
            m = 0
            for p, discard_idxs, choice_idx in choices:
                if choice_idx is None:
                    logger(
                        "{} does not have enough cards to compete in this tie-breaker... "
                        "they will have to discard their whole hand.".format(p.name)
                    )
                    idxs = discard_idxs
                else:
                    logger("{} discards indexes {} and selects the {} at index {}.".format(
                        p.name, discard_idxs, p.hand[choice_idx], choice_idx
                    ))
                    idxs = discard_idxs + [choice_idx]
                for idx in sorted(idxs, reverse=True):
                    card = p.hand.pop(idx)
                    if idx == choice_idx:
                        if card > m:
                            m = card
                        pot.append((p, card))
                    carry_pot.append(card)
            winners = [p for p, card in pot if card == m]
            logger("Winners for the tie-breaker are {}.".format(
                ', '.join(w.name for w in winners)
            ))

        if len(winners) == 1:
            logger("{} wins: {}.".format(
                winners[0].name,
                ', '.join(map(str, carry_pot))
            ))
            winners[0].wins += carry_pot
            carry_pot = []
        else:
            logger("Nobody wins! The GM mocks your inability to break ties!")
            logger("Carrying to the next round's pot: {}".format(', '.join(map(str, carry_pot))))
        for p in players_in_round:
            if len(p.hand) == 0:
                logger("{} is out of cards in their hand...".format(p.name))
                if len(p.wins) == 0:
                    logger("{} is out of cards in their wins pile. That sucks!".format(p.name))
                else:
                    logger("{} gathers their wins pile with {} cards.".format(p.name, len(p.wins)))
                    p.hand = p.wins
                    p.wins = []
    if len(players_in_round) == 1:
        p = players_in_round[0]
        logger("Congrats to {}, the winner of the game!".format(p.name))
        return rnd, p
    logger("Looks like you screwed this one up... everyone is out and nobody won!")
    return rnd, None
```