“BLUFF” WITH AI

CS297 Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Class

CS 297

By

Tina Philip

May 2017

I. INTRODUCTION

The goal of this project is to build an AI that learns how to play Bluff. Bluff is a logistically simple multi-player, non-deterministic card game in which each player gets to make a decision in each turn. The decision is the number of cards to play and which cards to play. This decision is based on imperfect or hidden information from the partially-observed game state that evolves under uncertainty. The process of bluffing involves making an unexpected move and thus misleading one's opponent [1]. The strategic complexity in the game arises from the imperfect/hidden information. Imperfect information means that certain relevant details are withheld or not known to the players and the knowledge is not entirely reliable. Decision making under conditions of uncertainty is one of the fundamental research problems in Computer Science and much research is being done in this area. Bluff is an extremely suitable domain for handling problems with decision making under unreliable or incomplete information with its strategic complexity and well-defined parameters like the type of card to be played in each turn [2].

In our project we will use neural network trained with back-propagation to build our Bluff AI. The preliminary steps are explained in the following sections. Deliverable 1 is the Base classes for the game, Deliverable 2 is a Bluff program for human players, Deliverable 3 is a simple AI player and Deliverable 4 is a study of how Poker AI players are implemented.

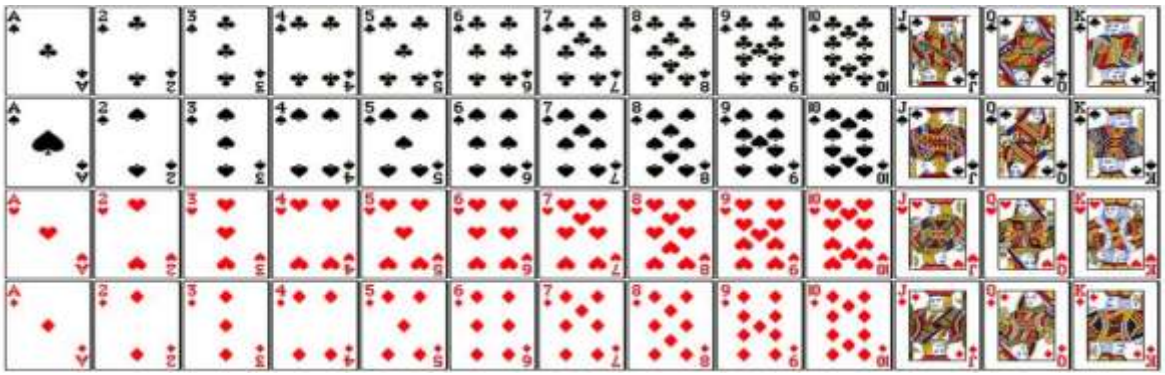

Fig. 1. A standard deck of 52 cards.

II. DELIVERABLE 1 – BASE CLASS FOR BLUFF GAME

The card game Bluff is generally called 'Cheat' in Britain,'BS' or 'I doubt it' in the USA and in India it is called Bluff. One standard pack of 52 cards as shown in Fig. 1 is used in the game and can be played by a maximum of two to eight players. Each player aims to get rid of all their cards to a discard pile as soon as they can. The discard pile starts empty and slowly increases as each player discards their cards. The rule specifies that each player plays the next higher rank than the previous player. The first player must discard Aces, the second player discards Twos, the next player Threes, and so on. After Tens come Jacks, then Queens, then Kings, then back to Aces, and so on. A maximum of four cards can be discarded in each turn, face down to the discard pile. Since cards are played face down, players have the option to bluff about the cards that they play. But if they are caught by another player, known as the challenger, the cards are exposed and the loser must pick up all the cards from the discard pile. One of the strategies in this game is to keep the opponents clueless whether you are playing the right cards or not.

The objective of this deliverable is to implement the Card class which is the base class for the game. The main logic starts from the Driver class which calls the Card class to establish the deck and also performs random permutation on it. The Driver class calls an `initialise()` method that establishes the parameters of the game such as the mode of the game and the number of players. The Deck as seen by the user is represented by their names as: Ace of Spades, Two of Spades, Three of Spades, ..., King of Diamonds. Internally, each card is assigned a number from 0 for Ace of Spades, 1 for Two of Spades and so on to 51 for King of Diamonds. The cards (or numbers) are permuted randomly to ensure none of the players get any advantage over the other. A technique called Randomize-In-Place [4] is used to generate a uniformly chosen random permutation with probability (1/n!). This method has a linear run

time. `Math.random()` method generates a number between 0 and 1, that is not a whole number and also is not 1. To get a number between 0 and 51, multiply `Math.random() * 51`. To make this value a whole number, or an an integer, apply `Math.floor` method which rounds it down to the nearest whole number like so: `Math.floor(Math.random() *51)`. To get the answer between 1 and 51, add 1 to the answer: `Math.floor(Math.random() * 51 + 1)`. This random value is then swapped in place with each of the values at locations from 51 to 0 in the array.

```
public int[] shuffle(){
    for(int i = 0; i< Deck.length; i++){
        cards[i] = i;
    }
    for(int i = 51; i >= 0; i--){
        int rand  = (int)(Math.floor(Math.random() * (i+1)));
        int temp  = cards[i];
        cards[i]  = cards[rand];
        cards[rand] = temp;
    }
    return cards;
}
```

Fig. 2.The `shuffle()` method shuffles the deck into a random order.

In the Player class, arrays for each player are created depending on the number of players established by the user at runtime. The cards in the deck are assigned to players so that each player has equal number of cards as shown in Fig. 3. It also contains functions to add cards to the players' array during the game when the player is caught bluffing as well as to remove cards to deal from a players' hand.

```
while (p <= players){
    Integer[] a = intArrayToIntegerArray(Arrays.copyOfRange(cardsArray, start, end));
    playersList.add(a);
    start = end;
    end += div;
    p++;
}
```

Fig. 3. Function to assign cards in the deck to the players.

III. DELIVERABLE 2 - A BLUFF PROGRAM FOR HUMAN PLAYERS

In this section we describe our implementation of Bluff that allows human players to play the game. Bluff is played by two to eight players and is decided by the user at runtime. A list of arrays is created for each player to hold their cards after shuffling the deck. The first step in the game is to choose the mode which is distinguished as Human-Computer, Computer-Computer or Human only as shown in Fig. 4. The Human only mode allows human players to play the game exclusively among them. The next step in this mode is to decide on the number of players and the rank of the card to be played in each turn is displayed starting with ACEs. The current player to deal cards in the game is found by the equation: i = (i+1) % players; where i starts from 0 and players equal to the total number of players in the game. In our program, Player 1 is shown his cards and asked to select the cards to deal. The player enters the card number in a CSV format and those cards are added to the discard pile. The game proceeds in two ways: either none of the other players call Bluff, in which case the turn goes for Player 2; or one of the players could call Bluff. In the case someone calls bluff, he becomes the challenger and the cards played by the player are compared with the actual cards to be played during that turn by `bluffVerifier`() method. The `getCardsOfPlayer`() method returns the cards of player. The `getActualCardsToBePlayed`() method returns the cards that should be played in the turn. If a player plays more than 4 cards in a turn, it must be a bluff, since there are only 4 cards of the same rank. The `bluffVerifier`() method checks if all the cards played by the player are from the same rank. If not, it returns a true verdict. If the verdict is true, the loser is the current player, otherwise the challenger loses. The loser has to pick up the cards in the discard pile.

```
Choose Game Mode (1-4)
1.Human-Computer          2.Computer-Computer
        3.Human solo      4.Exit
3
Enter the number of human players:
4

Current card to be played: ACEs

Turn for Player 1:

0 - NineofClubs          1 - FiveofDiamonds
2 - JackofClubs          3 - FourofHearts
4 - KingofClubs          5 - QueenofSpades
6 - FourofDiamonds       7 - Fiveofspades
8 - SevenofDiamonds      9 - SevenofHearts
10 - ThreeofSpades        11 - TenofDiamonds
12 - ThreeofHearts
Enter the no. of cards:
2
Enter card numbers in CSV format:
1,2

Player 2: Do you call Bluff? (Y/N)
N

Player 3: Do you call Bluff? (Y/N)
N

Player 4: Do you call Bluff? (Y/N)
N
```

Fig. 4. No-Bluff scenario.

The `addDiscardPileToPlayerCards`() method adds all the cards in the discard pile to the loser as in Fig. 5. We require that the player not lie about the number of cards that he has played in any round.

```
if(bluff){
    actualCardsToBePlayed = c.getActualCardsToBePlayed(current_card_to_play);
    boolean verdict = p.bluffVerifier(cardsOfPlayer,actualCardsToBePlayed);
    int loser;
    if(verdict){
        loser = currentPlayer;
        System.out.println("Caught! Player "+player+ " played wrong cards. The pile is now yours!");
    }else{
        loser = challenger-1;
        System.out.println("Player "+player+ " played correct cards. The pile goes to challenger.");
    }
    discardPile = r.addDiscardPileToPlayerCards(loser, r.getPlayersList().get(currentPlayer), discardPile);

    break;

} else {
    challenger = challenger +1;
}
```

Fig. 5. Code for the `bluffVerifier`() method.

After each round, the `findWinner`() method is called to check if any player has emptied his hand, if so he is declared the winner. The game continues till there is a winner.

IV. DELIVERABLE 3 – SIMPLE AI PLAYER

So far we have described a fully functional game that could be played by humans. Our next objective was to build a simple agent that plays the game. The goal was to make the agent play the cards of the correct rank for that turn if possible. In other words, the agent tries not to bluff at all. In the case that he does not have the card of the correct rank, he plays the first available card. The first step to play with the simple AI player is to choose the Human-Computer game mode which is mode 1. Then the number of human players and computer players are to be specified. Since there is just one simple AI player, the number of computer players is specified as 1 in Fig. 6.

```
Choose Game Mode (1-4)
1.Human-Computer          2.Computer-Computer
        3.Human solo      4.Exit
1
Enter the number of human players:
3
Enter the number of computer players:
1
```

Fig. 6. Choosing Human-Computer game mode.

As in the Human-only mode, the computer player is assigned cards, depending on the number of players in the game and gets its turn in each round. The agent knows all the rules of the game and is expected to play just like any truthful human player would. The agent knows the rank of the card to be played in that turn. It searches for a card of the same rank in the hand maintained in the array `computerPlayerCardsArray`. At the first occurrence of a card of matching rank from the array currentCardtoPlay, it stops the search and returns the card to deal. The `findCardsToPlay`() method takes the arrays `computerPlayerCardsArray` and `currentCardtoPlay` as parameters as shown in

Fig. 7. If there is a matching card in the hand, a boolean variable is set to true and card contains the number of the matching card. If not, retVal is set as the first card in hand denoted by `computerPlayerCardsArray`[0].

```
public static int findCardsToPlay(Integer[] computerPlayerCardsArray, int[] currentCardtoPlay) {
    // TODO Auto-generated method stub
    boolean contains = false;
    int retVal, card = 0;
    for(int j = 0; j < computerPlayerCardsArray.length; j++){
        for (int i : currentCardtoPlay) {
            if (i == computerPlayerCardsArray[j]) {
                contains = true;
                card = computerPlayerCardsArray[j];
                break;
            }
        }
    }
    if(!contains)
        retVal = computerPlayerCardsArray[0];
    else
        retVal = card;
    return retVal;
}
```

Fig. 7. The `findCardsToPlay`() looks for cards of a particular rank in the agent's hand.

V. DELIVERABLE 4 – A STUDY ON POKER AI PLAYERS

Bluff and Poker are similar in that, at any stage the players only have partial knowledge about the current state of the game. Incomplete or unreliable knowledge and risk management are factors that make Poker an interesting study for our project. The Computer Poker Research Group (CPRG) of the University of Alberta has contributed greatly towards the study of poker academically. They implemented a neural network for predicting the opponent's next move [2]. In [7] Koller states that just as there is an optimal move for every perfect information game, existence of optimal strategies can be proved if randomized strategies are allowed. Optimal strategy ensures that the opponent gains no advantage over the player even if his strategy is revealed. It can be shown that the theoretical maximum guaranteed profit from a given poker situation can be attained by bluffing, or calling a possible bluff, with a predetermined probability. The relative frequency of bluffing or calling a bluff is based only on the size of the bet in relation to the size of the pot. To ensure maximum profit, the move must be unpredictable.

Consider an example with 2 players in a game of pot-limit Draw poker where player 2 has called a flush with a draw of just one card against player 1 who has the best hand. Player 2 wins showdown (revealing of cards played) if she makes the flush, but loses otherwise. This situation is similar to when an opponent calls bluff and the player has to display the cards. If the cards are played right, the player is safe, but otherwise he loses and has to take the pile. The example further assumes that the probability of completing the flush is exactly 0.2 or one in five, since 5 cards make a flush. Game theoretic analysis can be used to calculate the number of bluffs and the numbers of calls that helps to build an optimal strategy. For each pair of frequencies, the overall expectation can be calculated as shown in Fig. 8. This is very

different from maximal strategies which aim to exploit weakness in an opponent's strategy.

	Player B Bluffing Ratio (BR) and Frequency (ABF)									
	0:1	1:4	**1:2**	3:4	1:1	5:4	3:2	2:1	3:1	4:1
CFr	0.00	0.05	**0.10**	0.15	0.20	0.25	0.30	0.40	0.60	0.80
0.00	.20	.25	**.30**	.35	.40	.45	.50	.60	.80	1.00
0.10	.22	.26	**.30**	.34	.38	.42	.46	.54	.70	.86
0.20	.24	.27	**.30**	.33	.36	.39	.42	.48	.60	.72
0.30	.26	.28	**.30**	.32	.34	.36	.38	.42	.50	.58
0.40	.28	.29	**.30**	.31	.32	.33	.34	.36	.40	.44
0.50	**.30**	**.30**	**.30**	**.30**	**.30**	**.30**	**.30**	**.30**	**.30**	**.30**
0.60	.32	.31	**.30**	.29	.28	.27	.26	.24	.20	.16
0.70	.34	.32	**.30**	.28	.26	.24	.22	.18	.10	.02
0.80	.36	.33	**.30**	.27	.24	.21	.18	.12	.00	-.12
0.90	.38	.34	**.30**	.26	.22	.18	.14	.06	-.10	-.26
1.00	.40	.35	**.30**	.25	.20	.15	.10	.00	-.20	-.40

Fig. 8. Expected values for a Four Flush Draw: Bluffing vs. Calling Frequencies

Currently we pursue the optimal strategy which can be used in Bluff as well, where we determine the probabilities of each card being in a particular hand. We know with certainty which cards are in our hand and which cards have been played. We determine the probability of all other cards based on cards that have been played. To start with, we could just use a naive and equal probability that a card is in some player's hand. If there are four players, each player have a probability of 1/39 or 0.025 to have a card, except for those cards which are held by the player himself with a probability of 1. Player 2 can call bluff if a threshold is met that a certain player does not have that card. A table can be maintained as shown in Fig 8 to calculate the probabilities. Whenever a player is caught, we gain insight to the cards he gets. We know that the loser will from now on possess all the cards in the discard pile with a probability of one. This discard pile has the cards that we played in previous turns and we can also keep a count on the number of cards in the discard pile.

VI. CONCLUSION

This semester, I had four objectives for my CS 297 project:

To develop the classes to play Bluff, to code a human only game of Bluff, to code a simple AI player and to decide on strategies to be explored in CS 298. All of these objectives have been achieved. My next goal is to build, an Intelligent Agent. The Intelligent Agent should be able to observe the opponents and adapt the strategy of the game for each. The agent should keep the information of all the movements made by the opponents throughout the game and use it to his advantage. The agent makes each move based on an optimal strategy and sometimes allow for some calculated risk.

The Heuristic approach as described by Billings [2] would be an effective strategy to achieve our objective for CS 298 which is to build an intelligent agent that watches every card that goes through the hand, calling bluff when a play is not possible due to where the cards are. This agent plays truth when able, but lies intelligently using the cards if it benefits. Another approach is to use cards to bluff that the agent is sure not to use any time in the near future. It can also try to call bluff on other players, since almost always the last card never fits. It can do this by keeping track of the number of cards of other players. Multiple agents could be created and played against each other to identify the best approach.

Many other approaches could be viable for producing strong algorithms, such as genetic algorithms or neural nets, the effects of which are yet to be identified, and could be the future research scope of this project.

REFERENCES

[1] Hurwitz, Evan, and Tshilidzi Marwala. "Learning to bluff." *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. IEEE, 2007. Available: https://pdfs.semanticscholar.org/ff49/bcf422168c6bfe4f115f02d098ad7bf49065.pdf

[2] Darse Billings. "Algorithms and Assessment in Computer Poker, " Ph.D. Dissertation, 2006, University of Alberta, Edmonton, Alta., Canada. AAINR22991.

[3] Russell, Stuart, Peter Norvig, and Artificial Intelligence. "A modern approach."*Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995): 27. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.259.8854&rep=rep1&type=pdf

[4] Pollett, C. (2015, Feb 2), *Random Permutations, the Birthday Problem, Ball and Bins Arguments.* [Powerpoint slides]. Retrieved from: http://www.cs.sjsu.edu/faculty/pollett/255.1.15s/Lec02022015.html#(1)

[5] Eastaugh, B. (2014, Feb 8). *The Mathematics of Bluffing* [Blog post]. Retrieved from https://ibmathsresources.com/2014/02/08/the-mathematics-of-bluffing/

[6 Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N & Bowling, M. (2017). *DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker*. arXiv preprint arXiv:1701.01724.

[7] Koller, Daphne, and Avi Pfeffer. "*Generating and solving imperfect information games*." *IJCAI*. 1995. Available: https://ai.stanford.edu/~koller/Papers/Koller+Pfeffer:IJCAI95.pdf