Draft Specification : Bayesian Network for Texas Hold'Em Poker

Group Members: Shanyi Gu (shanyigu@college.harvard.edu), Lane Erickson (lerickson@college.harvard.edu), Dana Katzenelson (dkatzenelson@college.harvard.edu), Robbie Burr Eginton (eginton@college.harvard.edu)

Brief Overview

We will implement a Texas Hold'Em game in OCaml with an interface for a human player and a Bayesian network controlling the actions of a computer player. We will implement functions that mimic the basic components of a poker game (dealing cards, betting procedures), allow the user choice over their actions while (ideally) providing them with useful statistical information. We would also have (ideally) a "wrapping" function that modifies the computer's behavior based upon past user actions.

From the user's perspective, running the program opens up a series of prompts, starts the user and the computer off at some amount of money in integer form, shows the user his/her first hand, and directly asks the user for each bet, etc. This would mean that, on the user level, there would be no function calls. This would mean also that gameplay would be restricted to two agents, the user and the computer.

Feature List

- Bayesian network that generates the computer player's moves upon being passed information about the game and making appropriate calculations
    - Implementation similar to what is described in http://ats.cs.ut.ee/u/kt/hw/bayesnets-poker/bayesnets-poker.pdf, http://www.csse.monash.edu.au/bai/poker/2006paper.pdf
- Functions that mimic the progression of a poker game (draw hands, flop, turn, river; choose best hand at end of the game for each player)
- User interface that shows user cards (and poker probabilities) and takes their bets
- Functions that provide information on poker probabilities at points within the game (most probable hands & best possible hands with associated probabilities, probability of winning the game)

- Wrapping function that modifies the Bayesian network based on information from past games

Draft Technical Specification

type card_val = 2 | 3 | ... | J | Q | K | A

val int_of_card_val : card_val -> int

(where J = 11, etc)

type suit = Hearts | Diamonds | Clubs | Spades

type card = card_val * suit

val suit_of_card : card -> suit

type group
(A group of cards, implemented as a tree, list, or other structure.)

type hand_type = High of int | Pair of int | Three of int | Full house of int*int | Straight of int (or int*int*int*int*int, but straights are defined by high card) | Four of int | Flush of suit | StraightFlush of suit*int (forgive me if I forgot anything)
(This hand_type definition will allow us to more easily tell the user what they have/what is possible.)

val hand_type_of_hand : card group -> hand_type

type move = Fold | Call | Raise

type comparison = Better | Worse | Equal

val compare : card group -> card group -> comparison
(e.g., compare [straight] [pair] -> Better)

val draw : card group -> card
(Draws from deck)

val generate_hands : card group -> (card group) list
(Give it the deck and it draws two cards for the user and two for the computer, and passes

back [[user hand];[computer hand];[remaining cards]]. Following this function call, the current dealer puts down 1 and the other can put down 2 or more (and should always choose 2) before four rounds of full betting proceed. After that, if no one has folded, the flop would be called. Although you could call the flop before betting and just keep its cards around, not accessed, there's no reason to.)

val generate_flop : card group -> (card group) list
(Given the deck (following generate_hands), passes back a three card group ("face up") and a one card group ("face down") before the non-dealing player can call, place a bet in excess of or equal to 2, and then the dealing player may act in the same fashion)

val generate_turn_river: (card group) list -> (card group) list
(Same as flop, but takes [deck; face up; face down] and returns the same format but with one card dealt to the face up list and one to face down group. Betting works the same as flop but with 4 as the min bet. It is called twice, once for the Turn and once for the River, with a round of betting after each.)

val choose_best_hand: (card group) list -> card group
(In Texas Hold'Em, the best hand is chosen out of the five face up cards and the two cards in one's hand. This function should accept a list of card groups and give back the best five-card hand. It will, in practice, only be passed a [2-card hand;5-card face-up group].)

val most_probable_hands (card group) list -> int -> move
(Displays to the user a number of most probable hands, with associated probabilities.)

val best_possible_hands (card group) list -> int -> move
(Displays to the user a number of best possible hands, with associated probabilities.)

prob_win : card group -> card group -> float
(Takes in the user's hand and the cards currently on the table, returns probability that user is (strictly) beating opponent.)

val generate_move: (card group) list -> move
(Passed whatever lists need to be passed the network, returns the computer's move.)

val modify_network: game list -> network
(No need for it to be in sig, but should be in struct when finished)

"What is next?"

We will learn the rules and statistics of poker and research how to best use a Bayesian network to implement them. Though the bulk of this project will be done in OCaml using modules, we will need to determine how to implement a user-friendly interface that prompts the user without requiring commands and function names to be entered. (One possibility for this is the object oriented aspects of OCaml.) We will also determine the best way to divide the work for this project among 4 people.