BAYESIAN POKERBOT FINAL WRITEUP

Dana Katzenelson, Lane Erickson, Shanyi Gu, and Robbie Eginton

May 5, 2013

Part I
Reflection Questions

1 How good was your original planning? How did your milestones go?

Our original planning significantly under-budgeted planning and design time. Actually implementing the algorithm required much more rigorous testing and debugging than any problem set but was relatively straightforward. Understanding how Bayesian inference worked and how to set up the Bayesian network was the part of this project that really took time. For instance, we initially implemented the network as a literal graph with nodes before realizing that the diagram was a way of representing tables, values, and functions. We also spent a lot of time working with the Tretyakov source before realizing that its implementation was faulty and demonstrated a lack of proper technical understanding. Other sources were littered with incorrect statements such as that P(hand|actions) = P(actions|hand) that were easy to miss at first read-through but invalidated key assertions.

Navigating through all of these sources and reconciling with their inconsistencies and mistakes was extraordinarily valuable in clarifying our technical understanding of Bayesian networks, but it also took an inordinate amount of time. Because of this, we realized at around the first checkpoint that our original functional and technical specifications were almost entirely misguided. Because of this, our first graph implementation had to be basically scrapped and rewritten for the second checkpoint, leading an intensely heavy workload in our second week that we did not originally plan for.

2 What was your experience with design, interfaces, languages, systems, testing, etc.?

See Part N for a detailed description of what we implemented and how we made our design choices. Addendum on testing: the abstraction evaluation module PokerEval required the most meticulous and comprehensive testing, totaling around 300 lines of asserts with many more ad-hoc tests in the command line. Testing the Bayesian network, its computational inputs and the controller/interface was tricky in that it was often difficult to pinpoint the source of each problem; this was done with copious usage of print functions and group powwows.

What surprises, pleasant or otherwise, did you encounter on the way?

The only real surprise was the necessity of re-implementing much of the network for the second checkpoint. We didn't realize just how flawed our original understanding of Bayesian networks was.

3 What choices did you make that worked out well or badly?

- Our choice to go ahead and implement the functional skeleton of a network despite not being completely confident in our understanding was somewhat ill-advised on hindsight, but it might have been necessary to fail hard at first to understand what to do the second time around.
- Our choice to implement poker abstractions and evaluation functions ourselves (see the gigantic PokerEval module) was extremely time consuming but extremely valuable, as we were able to obtain much more granularity and flexibility than if we tried to import the standard poker evaluation module written in C.

4 What would you like to do if there were more time?

- Learning: we would like our poker bot to be able to use previously played games to update its model of opponent strategy (dynamic opponent modeling instead of static opponent modeling as described in the Nicholson paper).
- Betting: we would like to be able to bet in increments of larger than one and to set limits on the amount that can be bet.

- Series of actions: currently, our poker bot only takes into account one previous action instead of a string of previous actions on the part of the human player. This is evident in raising: when the computer player bets and the human player raises in response, the computer player will keep raising until the human player stops raising and calls to end the round. The computer player is not properly taking into account how multiple raises on the part of the human player demonstrate a stronger pocket hand than a single raise.
- Multi-player interactions: adapting the bot to play against more than one human player.

5 How would you do things differently next time?

We would focus a lot more on clarifying our understanding of Bayesian networks from the very beginning instead of jumping into the coding immediately.

6 What was each group member's contribution to the project?

Dana was in charge of setting up and implementing the core Bayesian network and the utility function. Shanyi generated the conditional probabilities required for the different nodes of the network and wrote the opponent modeling function. Lane wrote the poker abstractions module for generating and sorting poker hands into 44 unique abstractions. Robbie wrote the poker controller and interface and various helper functions. Everyone did a lot of talking to figure out how to design the project together.

7 What is the most important thing you learned from the project?

- The value of human friendship. If we didn't like each other and get along well, this project would have been far more frustrating than it was, both in the design phases and the debugging phase.
- Hofstadter's Law / the value of planning for roadblocks. We did not expect to have to re-implement so much of the network for the second checkpoint, making for a very crushing workload that week. We will strive to plan for unanticipated time-consuming problems in the future.

Part II

Design Documentation

8 Main Feature: Bayesian Decision Network in PokaBayes.ml

PokaBayes.ml uses a Bayesian decision network to play poker with type action = Fold | Call | Raise | Check | Bet | NoAct (where NoAct means no prior action this game).

Our Bayesian Network is set up as an adaptation of the network displayed in the Nicholson paper, where Round, BPP_Past_Action, and Pot are single values, BPP_Final is a probability table, OPP_Final, BPP_Current, Board, and OPP_Action are generated Conditional Probability Tables, and BPP_Win, Winnings, and BPP_Next_Action are calculated from known information each turn.

A Bayesian network expresses the probabilistic relationship between causes (expressed by random variables) and effects (called evidence variables). When the causes are known, one can traverse the network to determine the probability of an event. When only the evidence variables are known, one can use Bayesian inference to calculate the posterior probability of the causes of these events.

A Bayesian decision network uses this information to calculate the utility of each possible outcome and select the outcome with the highest utility. Our utility function, a combination of the one expressed in the paper linked above and the one described in http://www.cbloom.com/poker/book/103_bayes.html, calculates the expected value of each legal game move by, for each legal computer player move:

$$\mathbf{Utility}(Act) =$$

$$P(win) * \left(Pot + Cost(Act) + \sum_{OppAct} (P(OppAct) * Cost(OppAct)) \right) +$$

$$\frac{P(tie)}{2} * \left(Pot + Cost(Act) + \sum_{OppAct} (P(OppAct) * Cost(OppAct)) \right) +$$

$$P(lose) * \sum (-Cost(Act))$$

Where $Cost(Act)$ is fixed (bets and raises are restricted to one unit each for simplicity). $P(OppAct)$ is calculated by using Bayesian inference to calculate the probabilities of the human's current hand given their past action, the current board (cards available to all in the center, may be empty), the computer's past action, the probabilistic relationship between human's current hand and human's final hand, and the probabilistic relationship between human's final hand and computer's final hand (once we have used Bayesian inference to calculate the probabilities of the computer's final hand given the computer's current hand). Then, once we have the probabilities of their current hand, we use those as weights and traverse the Bayesian network forward to calculate the probability of the opponent's future action given their current hand, the board, and the computer's past action. $P(win)$, $P(tie)$, and $P(lose)$ are all calculated as follows: first we use Bayesian inference to calculate the probabilities of the computer having each possible final hand given the computer's current hand. Then, we use Bayesian inference to calculate the probabilities of the opponent's final hand by first inferring their current hand (as described above) and then use those values as weights to infer their final hand given their current hand, the round, and the probabilistic relationship between the computer's final hand and the inferred human final hand. For each possible computer final hand, we sum the probabilities that the opponent's hand is better, equal, or worse (respectively) and multiply the sums by the probability that the computer has that final hand. We then sum these probabilities to give the probabilities of loss, tie, and win, respectively.

The computer knows which action to perform by extracting the action with the highest utility.

9 Supporting Feature: Conditional Probability Tables in Probs.ml and Bruteforce.ml

The Bayesian network does inference on conditional probability tables (hereafter CPTs). Due to the enormous potential size of these CPTs, we decided to use handtype abstractions to categorize poker hands (see next sub-section for details). As for data structure, we experimented with storing the data in hashtables or tree structures, but once we decided to use the abstractions it was simpler to store probabilities not as one single structure but as several lists of lists. As for generating the probabilities, our choices were to obtain them using combinatorial math or to brute force them by generating cards to simulate an extremely high number of games and then extracting probabilities from these. We chose the second method because we set up our decision network so as to require probabilities that would be very complex and finicky to calculate mathematically. All the CPTs were brute-forced (see Bruteforce.ml) using our card generation and card sorting / abstraction functions (described in next subsection), with the exception of the table involving predicting opponent future actions using opponent past actions, board handtype and predicted opponent current handtype. This was generated using several piecewise mathematical functions which were written to model the poker strategy of a medium skilled amateur player (see CPT_I_Love_You.ml).

10 Supporting Feature: Handtype Abstractions in PokerEval.ml

There are various programs available to compare poker hands, and to place 7-card hands into certain abstractions. However, for the purposes of this project, we needed a module which could: (a) compare 7-card hands to decide wins, losses, and ties at the end of a round; (b) place hands of size 2 through 7 into the same set of hand abstractions; (c) interface with the rest of our code; (d) handle card-drawing, representing cards/hands as strings, and other deck/card operations; and (e) be used to generate large CPTs. In addition, the granularity of these abstractions is extremely crucial, especially in the earlier rounds of the game. No available program or module existed that came close to meeting these prerequisites, so we constructed the necessary module ourselves (see PokerEval.ml.)

PokerEval has several levels of abstraction granularity available, each building on the smaller levels. The simplest and most naïve sorts hands into any of the nine conventional hand types. Since two-card hands could only be a pair or a busted hand type, this was simply not granular enough. Building upon the type "hand" is "expandedhand," which divides some of the original nine types according to their high cards, and then "strhand," which wraps around "expandedhand" and signifies either the presence of a significant partial straight or its absence, and "broadhand," which does the same but with partial flushes. In addition, busted and pair hands were sorted by high cards, making for an overall 44-type abstraction. The ranking function provides either the 9-type or the 44-type hand abstraction depending upon user preference.

Our choice to sort by partial straights and flushes was unconventional. The Nicholson paper asserted that extra nodes would be necessary to access this information without compromising type invariants, which was why it was not included within their 25 abstractions. We dealt with some design considerations with the partials: for example, revealing the presence of a flush and a straight might signify a straight flush or just a straight, and filtering the straights from the straight flushes made for quite a hassle. Another necessity is that cards of a better rank beat cards of a lower rank in the final hand, for which partials present a problem. A partial straight in a hand is better than an equivalent hand type without a partial straight, but it is not relevant in the final round. However, we dealt with these intra-round ranking inconsistencies without extra nodes by selectively using our wrapping functions (expandedhand and broadhand) to ignore partial straights and flushes only in the final 7-card hands.

11 Supporting Feature: Game Controller and OOP Interface Wrappers

The "controller" is the system that handles user IO, manages the sequence of turns, rounds, and games, enforces legal play, keeps track of the state of the game, and feeds information about what's going on (in very different ways) to both the human player and the Bayesian AI. The PokaBayes module implements the AI's decision-making process, and the PokerEval module facilitates the card-related rules of Poker (i.e. generating, classifying, and comparing hands), and the Probs module gives the AI a sort of "experience" with poker in practice, then the controller is in a sense the environment through which the game is played. The controller system is split into two parts. Controlla.ml defines the flow of the program and the flow of a game. Players.ml defines object classes for the human and the computer player which conform to a single class type player_t and which hold the player-specific logic relevant to either IO or the interactions with the AI. This will become more clear later in this summary.

11.1 Game Logic in Controlla.ml

Controlla.ml implements the top-level flow of the program and the turn-round-game-flow of the games. It defines a group of IO functions at the top. Most of these are specialized functions designed to handle exactly one interaction – i.e. a function to greet the user, a function to ask if the user wants to go first or second on the first game. One of them is a general function (get_opt) which handles getting input with two possible responses. The function select_winner does IO and also calls PokerEval functions to determine who won.

It defines two outer functions games and rounds, and a function inside rounds, turns. These are three three epicyclic loops of the program. Each call of games is a new game, of rounds a new round, of turns a new turn. rounds and turns are player invariant, while game is not. This means that there is nothing in rounds or turns which makes reference to which player is which – rounds and turns treat both the human and the computer the same way. games does not do this because it must be able to handle which player will go first and also must input the players to select_winner with the computer first and the human second.

In turns/rounds, there are essentially 5 variables that flow from iteration to iteration. These are:

1. the cards dealt so far
2. the two players (which alternate positions in turns but maintain their positions in rounds, since poker is turn-based but the same player starts each round)
3. the player who made the last move (not predictable from the positions of the players across round boundaries)
4. whether this is the first turn of a round
5. and the round number.

In essence, Controlla.ml is about rounds/turns. It implements a clean, not-particularly kludgy flow of the game, which uses the player_t abstraction in

Players.ml to be treat both human and AI players exactly the same. The only kludge I know of is an information flow not reflected in Controlla: on the first turn of a round, the computer player queries the human player for its previous action directly, instead of receiving this from the main controller flow. For specifics, see the comments.

11.2 OOP Interface Wrappers and Individual Player Logic in Players.ml

The oppl (human) and bppl (computer) objects both inherit from a generic player object, and all implement the same type, player_t. They actually override relatively little code, although both have some small number of private methods as well. bppl is parameterized with an instance of the opp object as well because it needs to call certain methods on it.

These objects maintain a record of the current pot, round number, cards on board, and of the cards in their respective hands and of their last action.

On a turn that is not the first turn of a round, player#turn will be called, and the action just taken by the other player will be passed in. The turn function adjusts the pot by an appropriate amount (i.e. by 1 for a bet, 2 for a raise, etc) and then calls self#get_action. get_action is different depending on which player it is. But both players' get_actions end with a call to do_action passing in the just-acquired action, which sets the player's last_action to the just acquired and adjusts the pot according to which action it was. do_action returns the new action, which returns all the way out to turns, the function that called player#turn.

oppl's get_action involves calling print_game_prompt, which assembles the current state of the game and presents it to the user. It then takes a validated input – one of three options depending on what the opponent's previous action was – which is what action the human is taking.

bppl's get_action involves calling PokaBayes.turn and passing in an array of arguments specifying the hand-type of the computer's hand (including the cards on the board and the cards in the hand), the round number, the computer's last action (unless that action was in a different round, in which case it passes in NoAct), the opponent's last action (even if that action was in a different round, the handtype of the cards on the board alone, and the amount of the pot. This action is then passed back to turns.

On a turn that is the first turn of a round, player#first_turn is called instead of player#turn. This will eventually call turn, but has to do some business first. In fact the order of calls on the first turn of a round is first_turn, turn1r, turn, get_action, do_action. first_turn is inherited, but turn1r is overridden by both players. first_turn prints some information about the state of the game and notifies the human that the round is over with an explanation as to what move and whose move ended the last round. Then first_turn prints a notification that the new round has started and says who goes first. Then first_turn calls turn1r. In oppl, turn1r always calls self#turn passing in PokaBayes.NoAct. This is because the human has already been notified of the last event of the preceding round, and so that the game-prompt does not print out said last action as though it had happened in this round. In bppl, turn1r passes the last act of the human player to self#turn, regardless of whether or not the human was the last one to go (remember that across round boundaries turn order is reset and so someone may in a sense go twice in a row). This is because of the requirements of the PokaBayes system, and is the kludge mentioned above.

By the way, the computer object also maintains a special instance variable "internal_last_action" which should be the same as its actual last action unless this is the first turn of a new round, in which case it will be equal to PokaBayes.NoAct. Updating this to NoAct is the job of player#more_new_round, called from player#new_round. Updating this on each new action was the job of the computer's get_action, since that is the only thing after the action has been acquired that is specific to each player (do_action is shared).

Part III
Guide to Using, Testing, and Understanding the Code

12 Instructions for Use

Follow the instructions that will appear on the screen. Enter prompted responses in the command line. Notes: when prompted for a move, type the move in the command line ("bet" "call" "check" "raise" or "fold"). You do not need to type it in all caps. The first time you play the game, the controller will let you select whether you want to play first. For each following iteration of the game, the controller will alternate the order of play between you and the computer player.

At the end of each iteration of the game, the interface will display who has won, what cards they won with and the size of the pot.

13 Tips for Testing

1. You may notice that our lists of utilities of future computer actions include the value "-nan". This is deliberate - you may notice that we set this up in the variable "imp" within the PokaBayes.turn function. Since "-nan" is the smallest float value (according to the Ocaml documentation it will compare as "-nan" < any_float), we are using it to represent illegal moves in our utility function so that they will always have the lowest utility and will never be selected as attempted moves. (e.g. you can't check a raise).

14 Tips for Understanding the Source

1. Here are some of the abbreviations and idioms that we frequently use in code:

   (a) CPT: Conditional Probability Table

   (b) PT: Probability Table

   (c) BPP: from "Bayesian Poker Player" – computer

   (d) OPP: from "Opponent" – human

   (e) Handtype/Hand type: either the actual hand type (partial straight, two pair, etc) or the integer representation thereof, depending on context

   (f) p1: Player 1

   (g) p2: Player 2

   (h) curr: current, usually referring to handtype

   (i) fin: final, usually referring to handtype

   (j) NoAct: no prior action, used at beginning of rounds of play, especially beginning of game

   (k) inf: inference, refers to method of obtaining the table

   (l) BOPP: used for both bpp and opp

   (m) gen: generate, used in function names

   (n) na: NoAct

   (o) fwd: forward, non-inference, refers to method of obtaining a table

2. You can activate "debug" messages outputted on stderr if you want. This will allow you to look inside the code. Instructions for doing this are in the README. If you want to add more debug messages to see things we haven't already exposed, then you can use print_debug in Controlla.ml and Pokabayes.ml, and within the objects in Players.ml, you can use self#print_debug. There is no print_debug in PokerEval.ml, but you can pretty easily add it along the lines of the definitions in Controlla and Pokabayes.

References

[1] Tretyakov and Kamm: http://ats.cs.ut.ee/u/kt/hw/bayesnets-poker/bayesnets-poker.pdf

[2] Nicholson, Korb and Boulton: http://www.csse.monash.edu.au/bai/poker/2006paper.pdf