DEEP GOLD: A Texas Hold'Em Simulator

CSE331: Data Structures
University of Notre Dame
Fall 2004

Garrett Britten (gbritten@nd.edu) - Daniel Mack (dmack@nd.edu) - Jared Sylvester (jsylvest@nd.edu)

Brian Bien (bbien@nd.edu)

1 ABSTRACT

The following document outlines our goal to implement a Texas Hold 'Em poker engine that can simulate the game of Texas Hold 'Em and make intelligent decisions at the playing table for the end user. This project makes use of several data structures, algorithms, and statistical data to allow the calculation of the odds at a much higher level and with much greater efficiency than the human mind. However, this poker engine is not able to play with the emotion of a human player and thus unable to read human reaction or simulate the art of a poker bluff. The simulation is coded in C++ and runs behind a user-interface that is developed within the Qt API.

KEYWORDS: poker, simulation, data structures, C++, Qt

2 INTRODUCTION

It is undeniable that Texas Hold 'Em poker is the most popular card game in the world at this time. From professionals to amateurs, this game holds the risk of losing it all and the excitement of winning big all at the same time for any player in the game. However, as with any game, there is a right and wrong way to play, and our project is designed to help any aspiring World Series of Poker competitor become an instant contender. When dealing with a deck of cards it is about probabilities, and the player who can most quickly determine the probability of a win with the cards dealt to him or her is able to make the decisions required to win the hand. Our application is designed to run on a laptop that is at the playing table with the end user. As a game of poker progresses, the user is able to input information into the user interface and ask the computer to simulate the current state of the game and make a recommendation as to what the user should do. Of course the user has the freedom to make his or her own decision and face the consequences. This application was designed using the C++ programming language the the Qt GUI developer. It is a multi-platform application supported on Microsoft Windows, Mac OS X, and Linux.

3 THE PROBLEM

Poker, more specifically Texas Hold'em, is a game based around imperfect knowledge, and as such, is hard to adapt an AI (Artificial Intelligence) to find the best course of action. Also, with multiple players and options for each player, an entire game tree can span through thousands upon thousands of nodes thus making a quick solution hard to find. It was at this point that we decided to tackle Texas Hold'em and create an application that would take user input of the game and an AI that would, on a button click, take in the multiple variables of the game and determine the best action to take.

4 THE SOLUTION

We wanted Deep Gold to be a program operated similar to Deep Blue: we wanted it to be able to play against human opposition through a proxy human that would act on its behalf at the game table. In the realm of poker, this means that the computer operator would be seated at the table with the human opponents. The

operator tells Deep Gold what is happening in the game (what cards are dealt, who folds, etc) and Deep Gold then makes a decision and tells its operator to make an action at the table for the other players to see.

This mode of operation also allows Deep Blue to act as an advisor for a human player. If a player were to take their laptop with them to the table, and keep Deep Gold updated on the state of the game, it would be able to advise the player as to appropriate moves.

Our GUI was designed with this type of operation in mind. The user must be able to inform the computer quickly and easily of all actions in the "real" game, so as to keep up with the sometimes fast-paced actions. Furthermore, the AI must be able to generate a decision quickly enough so as not to bog down the flow of the game.

4.1 Back End

AI We make use of two independent processes for decision making within the AI, calling the appropriate one based on the current game context. Before the flop (that is, the first three community cards that are dealt) we use an expert system to make decisions. The appropriate action at this stage of the game has been extensively studied in poker literature, and it was relatively simple to develop a method to determine the appropriate action based on the strength of the computer's cards, its position at the table and the relative toughness of the current game.

For the three rounds of betting which follow the flop, the appropriate decision is less clearly defined. In these situations, when there are no straight forward rules to follow, we rely on a simulation mechanism. Similar to Deep Blue, and many other game solving AIs, we seek to enumerate the possible outcomes given the current state of the game. Because poker is a game of imperfect information, enumerating every possible outcome is currently infeasible. In light of this, our strategy is to run a limited number simulated hands in order to keep the computation time reasonable.

Simulator The simulation process can be described in the following pseudo-code:

```
assume computer_decision is call
for( i = 0; i < 512; i++)
      deal_cards( deck )
      enumerate_branches(game_tree, current_state)
      EV_i = AVG( net winnings of each leaf )
EV_CALL = AVG( EV_0, EV_1, … EV_512)

assume computer_decision is raise
for( i = 0; i < 512; i++)
      deal_cards ( deck )
      enumerate_branches( game_tree, current_state )
      EV_i = AVG( net winnings of each leaf )
EV_RAISE = AVG( EV_0, EV_1, … EV_512)

EV_FOLD = 0.0;
```

Fig. 1. Psuedo-code

Currently, we simulate the hand, from the current point until the end, 512 times. We think 512 trials gives enough data to generate a reasonable decision while still being solvable in reasonable time. Each trial first deals out cards for all those that are unknown on the table. Cards are not dealt entirely randomly - opponents are more likely to be dealt stronger cards, since it is likely that they would have previously folded if they held weak cards.

The simulator seeks to figure out which of the possible actions (folding, betting or calling) have the highest expected value. The utility of folding is always zero, since it is impossible to gain or loose any more money when folding. (Prior wagers put in the pot are best treated as "sunken costs" and thus ignored.) The expected value of calling or raising is determined by setting the computer's current choice to that action, and then following the rest of the game tree through to the end of each branch. The branch's utility is positive, and equal to the pot size, if the computer would win the game represented at the end of the branch. The utility is negative, and equal to however much the computer would loose in wagers, if it would not win at the end of the branch.

Data Structures Each player in the game must store the relative strengths of any pair of hole cards. The computer must know their strengths in order to make well-informed pre-flop decisions. The human player objects store the hole card strengths so that the simulator can accurately deal the player cards which they are likely to be playing with. These strengths are stored in a map¡string, double¿ to allow fast access. We knew the data in these maps would have to be read far more often than written, and maps provide logarithmic access time. Insertion and removal was not a serious issue: since each map must store every possible hole card combination 1404 items, members would never be deleted or inserted (after initialization).

The second principle area where choice of data structures was important was within the simulator. Because the simulator was potentially solving hundreds of millions of branches in the game tree for every decisions it would make, it was critical that each individual trial be as efficient as possible.

One of the tasks the simulator must perform repeatedly is dealing cards to the remaining spots on the table and to the other players. To do this, cards that are already on the table must be removed from the "deck." This requires the container of cards which comprises the deck to be indexable with strings. (The method of representing a single card is a string which gives its face value and suit.) However, we must be able to draw a random card from this same deck. This requires being able to index the container with integers, since we can easily generate a random int.

We tried numerous times to develop a solution that would allow us to access a single container with either an integer or a string, but finally decided it would be simpler to just use two different containers. We implemented the deck with two maps which stored pointers to cards, one indexed with strings and one indexed with integers. The first allowed us to access a specific card if we knew what it was, and the second allowed to to draw a random card from the deck without knowing what card it would be.

Within the simulator itself, the current state (pot size, number of players, etc) of the simulated game must be stored. Since the many sub-branches of the tree would each require their own state, we wanted to keep the state objects as small as possible. To accomplish this goal, we used a list of the current players in the game. This allowed us to simply and quickly remove a player from the state if they folded, thus eliminating the overhead of having to continue storing that player's information as well as flags to see if they were still active. The list of active players allowed constant time removal of folded players. And since we only required access to on player at a time (the player whose turn it was to bet), there was no need to random access such as that found in a vector.

4.2 User Interface

Design The user interface was developed using the Qt API for both Windows XP and Linux. Qt for C++ is a very user friendly designer if you have some knowledge or resource to aid in the beginning. Our GUI was designed to be straight forward and clean for the end user. It was our goal to facilitate the input of information by the user as a game of poker progresses at the playing table. However, it was also important for this application to look nice a flow seamlessly as the user makes said inputs. This interface allows for user mistakes while also allowing for a level of freedom that a poker game may require. At the table, the action can be unpredictable at times, and the user will need the ability to account for this un predictability in real time. The UI is laid out on a 800x600 pixel window with the controls being placed in such a way that it is least confusing to the user. Several dialogs are also implemented to add options and functionality to the application. Most importantly, the deck of 52 cards is implemented as a matrix of buttons on a separate dialog window. These clickable card images allow the user to easily (pictorially) set the cards that one is dealt or set up the community cards. Once the UI is working at the highest level, it's true functionality must come from the underlying back-end which would be accessed through a gameplay class.

4.3 Connection

Getting the GUI to successfully communicate with the back-end was a non-trivial task, especially since different members of the group had different visions of the program behavior. The plan was that the game class would provide all the necessary accessible member functions for the communication to take place between the GUI and the back-end. When we began merging the front and back-ends, neither game.h nor the GUI had all of the functionality that was needed to control the flow of the game. The team member in charge of coordinating the merge decided that it would be best to create a struct to aid in the communication between the two layers. This approach was contentious, but proved successful, if a bit awkward, in the end.

5 CONCLUSIONS

We experienced numerous problems in coordinating the interface and the back-end which managed to knock us far off schedule. As a result, we had a very limited amount of time to test and debug the simulator. As of the due date, the simulator was not functioning entirely correctly, and we were forced to take it offline and use a much simpler method of post-flop decision making which only evaluates the computer's current hand strength. Because we were never able to implement a fully-functional simulator, it is impossible to determine its effectiveness. However, we are confident from our understanding of the existing literature on the problem that our solution would at least come close to meeting our original goal of being a competitive poker player.

6 FUTURE WORK

Obviously there is work to be done to meet our original design goal of a functional poker simulator. We feel that another two to three days would have been sufficient to resolve the problems. In addition to the simulator, the computer player can begin to capture more statistics form the game in order to achieve more specific opponent modeling. We designed our system from the beginning to be able to handle various opponent modeling techniques, so the framework is already in place. However, these were secondary goals and as such, were not completed. We feel that our design is flexible enough to allow many improvements to be made that would only become apparent with a greater understand of advanced poker strategy.

7 SCREENSHOTS

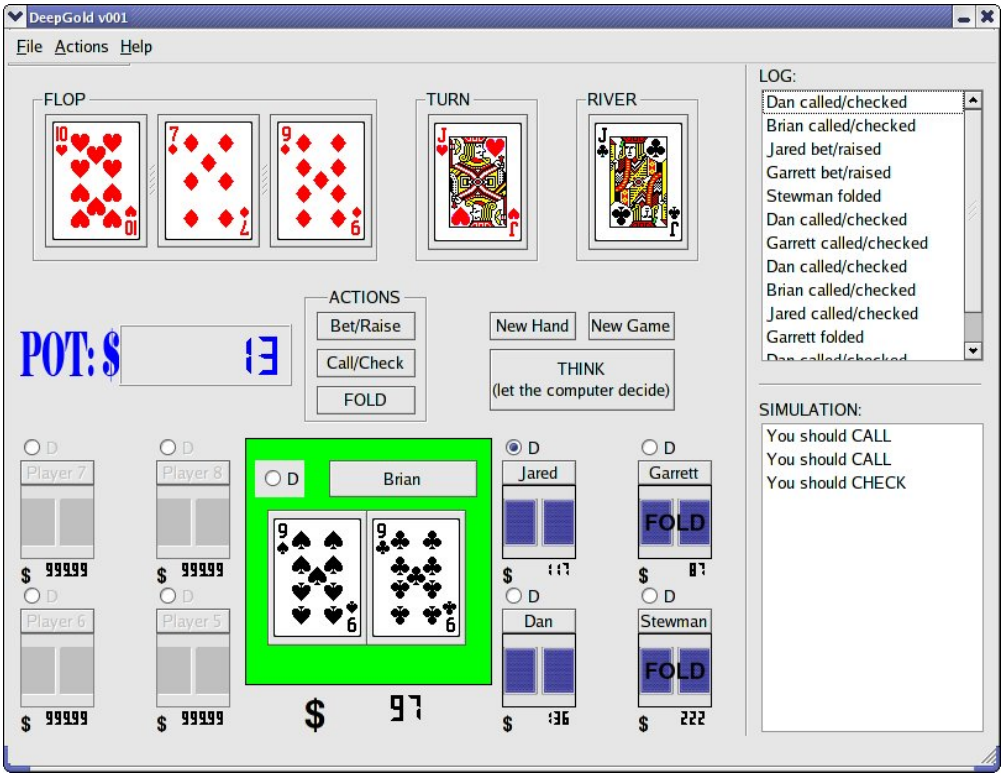

Fig. 2. Deep Gold

Figure 2 is the main window after the user has begun to play a hand of poker with other humans at a game table. Only five players are sitting at this table and and all of the community cards are showing. The last round of betting is beginning and the computer is suggesting a CHECK action for the user.

8 BIOGRAPHIES

Fig. 3. Garrett Britten

Garrett Britten returns as the GUI guy in another great Computer Science adventure with old friends Daniel Mack and Jared Sylvester. But wait...a new guest star has joined known as Brian Bien. Garrett is a Texan from a small farming community that nobody has ever heard of. Maybe someday he'll be famous a give Groom, TX a spot on the map...or maybe not!

Fig. 4. Daniel Mack

Daniel Mack is a native Nevadan thus intrinsically tieing him to the vice that is so common in the applications he chooses. A great communicator with absolutely no ambitions that he cares to share, our intrepid Nevadan finds himself at the crosshairs of history, but the real question remains, is he a crack shot? Daniel likes Macintoshes and the Federalist Papers.

Fig. 5. Jared Sylvester

Jared Sylvester was born and bred in suburban Maryland. The time he spent inside the beltway has given him a distrust of the Feds, over-priced coffee and soy products. He does have enormous respect for a cognitive reverie as well as respect for a simple meal of thin crust pizza and beer. Jared would like to pursue further study in the fields of Artificial Intelligence or Human-Computer Interaction or High-Seas Swashbuckling. Jared finds monkeys funny.

Fig. 6. Brian Bien

Brian Bien was born in Lansing, Michigan. He's lived in the beautiful state all his life, but would possibly like to live out west some day. His interests include hockey, chess, Hold'Em poker, object-oriented programming, and computer security. He would like to say more, but can't think of anything witty whatsoever, because he has spent too much time in front of the computer lately.

9 RESOURCES

Sklansky, David and Mason Malmuth. **Hold'em Poker:For Advanced Players.** Nevada: Creel Printing Co., 1999.

Findler, Nicholas. **Studies in Machine Cognition Using the Game of Poker.** State University of New York at Buffalo, 1977.

Findler, Nicholas, et al. **Heuristic Programmers and Their Gambling Machines.** State University of New York at Buffalo, 1974.

Billings, Darse, et al. **Using Probabilistic Knowledge and Simulation to Play Poker.** Ameri. Soc. for AI, 1999.

Schaeffer, Jonathan, et al. **Learning to Play Strong Poker.** University of Albert, Canada, 1999.

Qt Reference Documentation. http://doc.trolltech.com/3.2/. Accessed: April 2004.

Trolltech. **Qt 3.3.3 Whitepaper.** www.trolltech.com