Texas Hold'Em Group 03

79109P Antti Ahonen antti.ahonen@aalto.fi
79874V Mikko Salonen mikko.salonen@aalto.fi
58122B Mikko Närjänen mikko.narjanen@aalto.fi

Instructions for compiling and use

This program has been tested and compiled on Ubuntu 10.04.3 LTS. It should be compatible with Linux based systems. To compile this program all you need is the g++ compiler. We did not use any external libraries. Instructions:
1. Unzip the odds.zip to ~/holdem3/
2. Run g++ -std=c++0x -o holdem3 *.cc
3. Run the program by typing ./holdem3

You need to set the terminal character set to UTF-8 to see cards right. The program uses Linux terminal colors.

Our program is a Texas Hold'em simulator. It features a text-based user interface. Valid inputs are HELP, PLAY, QUIT, ODDS, RAISE, CALL, FOLD (type HELP in game for more information).

Program architechture (see appendix)

The game works by moving through States using the StateMachine. The States represent phases of the game, e.g. PreFlop. A state has three functions, Enter, Execute and Exit. The Enter function is used when moving to a state, for the example of PreFlop it forces players to pay the blinds and it deals the hand cards. The Execute function is the implementation of the game round, i.e. each player takes his turn. The Exit function serves to collect the pot and reset some values before moving to the next state. Both the StateMachine and the States themselves are singletons.

The Game class is basically a storage object wherein we store information about the players and the game. The Game class is a singleton. It keeps track of the dealer, the blinds, call and raise costs and the players.

Player classes represent human and AI players in the game. Common methods and data for the subclasses Human Player and AIPlayer are defined in the superclass Player.

There are a number of functions defined in HelperTools and the two most important ones are calculateHandValue and calculateOdds. HelperTools are used by Players and by the game logic. The AI algorithm uses them extensively in its decision making. In the game logic they are used to determine the winner.

Our game runs on a text-based user interface with command line inputs.

The reason we decided to implement the game logic using a StateMachine and GameStates is because it seemed logical (suggested by our assistant), this game being a turn based game with discrete stages. Our choice of using a vector to keep track of players wasn't the best, however at the stage of programming we understood this it was just easier to keep going instead of redesigning the whole thing.

Data structures and algorithms

Most of the data structures are vectors. The Card is a class containing the information about a card, that is color and value. The Deck class contains a vector of Card pointers. It has the ownership of the cards for memory management. It features two methods, one for shuffling the deck and the other for drawing a card from the deck.

As mentioned above, the game is a storage object. It stores the players in a vector, and the table cards in another vector. It keeps track of the pot, blinds, dealer and blind Ids. The game also owns the deck of cards. The methods that handle players are somewhat complex because of the need to handle players by Id. In retrospect we should have created our own PlayerList type of class for this.

There are two types of players in this game, Human and AI. They both inherit the Player class. The shared public methods in the Player class are different types of setters and getters for handling money and the players' status. The shared protected methods are raise, call and fold, which implement the respective actions in a poker game. The Player class defines a pure virtual method playTurn() that the subclasses implement.

The playTurn() implementation for the Human Player uses the GameUI to get input from the player. The playTurn() method for the AI class uses fuzzy logic to determine the best action to take. The AI uses two fuzzy logic variables handGoodnes and riskLevel. HandGoodness is based on the victory probability from the HelperTools, and the risk level is calculated from the number of players who have raised this turn and cost of Ais planned action in relation to the amount of money the AI player has. Both playTurn() methods use the superclass's raise, call and fold methods.

As explained under the previous heading, the States are classes that represent the stages of a poker game. They don't really store any information, they just fetch it from the Game and the Players.

Data structures used within the HelperTools are different variations of vectors. HelperTools contains two major algorithms used in the game. The most important is the calculateHandValue-method, which uses Moritz Hammer's 7-card DAG evaluator. More about this is in the references. This method reads the pregenerated DAG to memory, from where it traverses the DAG with 1-7 nodes. When it reaches its end, it gives a integer value of the given card-combination and its equivalence class. This information is stored in a vector.

The other major algorithm is the initOdds-method. It generates the wanted amount of Monte-Carlo simulations of games, from which the information about the cards and the winner of the game is stored in a two dimensional vector (separate two-dim vectors for basic odds from high card to straight flush and another two-dim vector for gathering the necessary information about the win statistics of the hands). This is used for generating the probability look-up tables. The algorithm for these simulations is not optimized enough and is not fast enough to generate accurate odds on demand. This is the reason for the developed write/read odds methods, which store and load the MC-simulations.

In general the reason we used vectors so much is because they're easy to work with and in the best case they offer the speed of arrays but are variable in size.

Known bugs

The odds for flush/straight flush are not 100% accurate because they are based on a Monte Carlo simulation which only takes into account the ranks of the cards player has in hand/table. We ran into a bug where the betting round got stuck in an infinite loop, but were unable to find the bug or even reproduce it. This it may still remain if you get unlucky* (*LATEST CHANGE 15.12.2011: This bug has now been fixed).

Task sharing and schedule

Task sharing went pretty much according to plan. Most of the time we coded together at Maari. Some work was done individually by group members. We had multiple meetings at Maari, at least once per week. Most of the work was done there. The schedule was accurate and we managed to follow it quite well. We didn't quite get the full working program by 30.11. as we had planned, but the main obstacles were solved at this point. Basic pieces for the game were done by the end of November as was planned. Excluding the odds, we had the game ready on 13th December. The generations of the odds tables proved a little problematic; working odds and a fully working game we had on 15.12.

The amount of work done by each group member was about 50-60 hours total. If we approximate the program took about 170 man hours to finish, it was divided mainly between the Game and GameStates (30%), Player classes (30%) and GameUI and HelperTools (40%). This includes testing and unit testing of which we did a lot. We estimate about half of the time spent done was testing. After the project plan we used about 2-3 hours for planning the StateMachine (game logic) and the UI. Learning to use the Monte Carlo simulation and figuring out the data structures for storing the probability information took Antti about 12 hours. The writing this project documentation took about 5 hours in the end.

Our scheduling and task sharing worked well because we worked closely together in form of meetings, where we discussed and planned implementations and also coded stuff. We are very happy with how things went. We didn't quite have the final program when we had planned, but other than that it was great.

Differences to the original plan

The only that differs a lot from the original plan is the addition of a StateMachine class and different GameStates that represent the phases of a Poker game as per recommendation by Petteri. Our plan was mostly good enough, but we should have thought more about the implementation of the game logic and how to calculate the probabilities and the values of hands.

References

Texas Hold’em rules
http://en.wikipedia.org/wiki/Texas_hold_%27em

Fuzzy logic wikipedia
http://en.wikipedia.org/wiki/Fuzzy_logic

Monte Carlo-method Wikipedia
http://en.wikipedia.org/wiki/Monte_Carlo_method

Moritz Hammer's 7-card DAG evaluator
http://www.pst.ifi.lmu.de/~hammer/poker/handeval.html

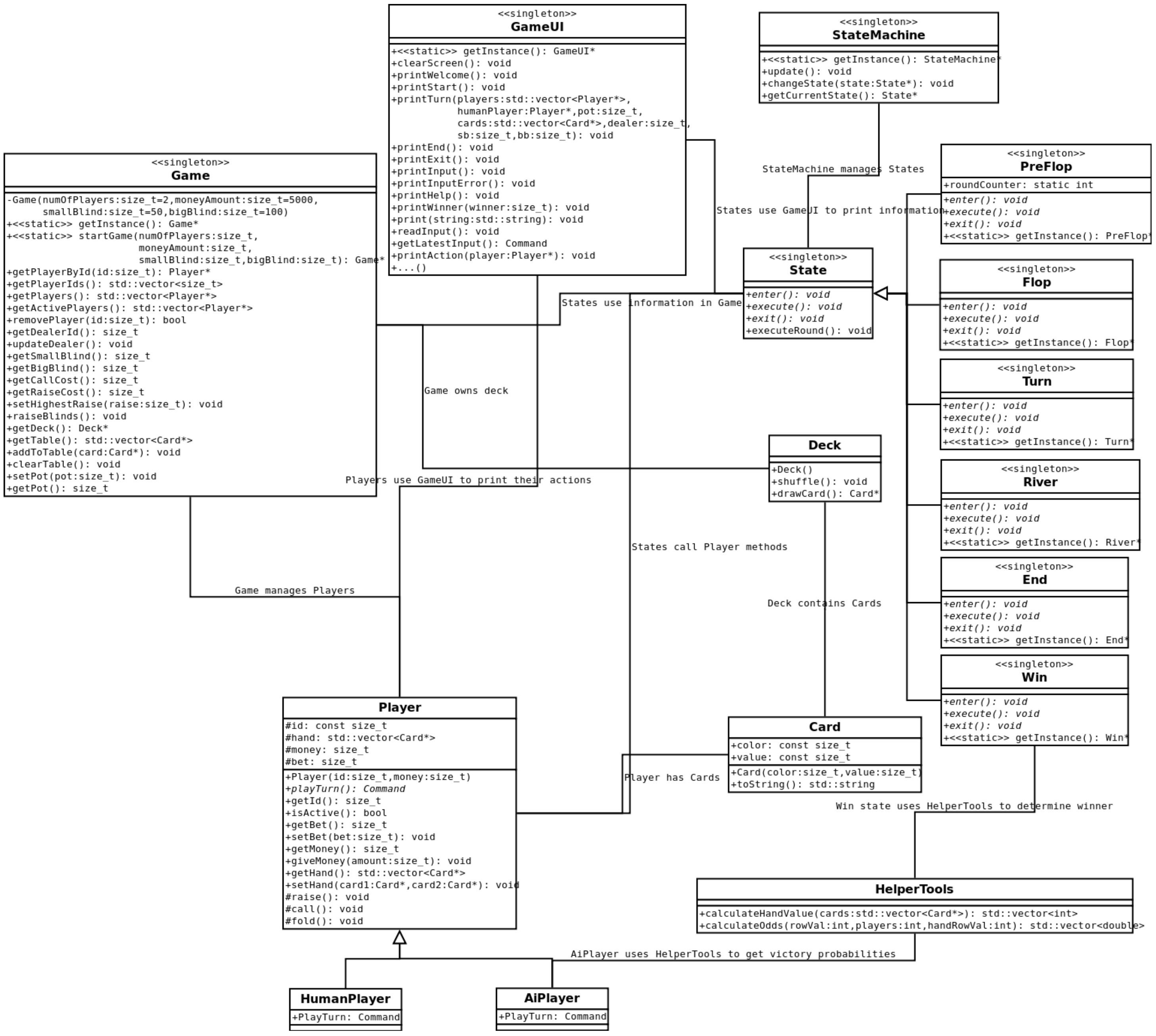
Pokeri-todennäköisyyslaskin for testing
http://fi.pokernews.com/poker-tools/poker-odds-calculator.htm

Example run of the program (beta)

```
kosh src 1103 % ./holdem_03
Welcome to the Texas Hold'Em game.
Type "HELP" for commands or just start playing by writing "PLAY"!
Input: HELP
The general commands are: PLAY, QUIT, and HELP. Game commands are: RAISE,
CALL,                                   FOLD and ODDS.
Game commands work only while playing.
Input: PLAY
Hi awesome HUMAN player. How many players do you want the game to have (2-
6)?
Please type 2,3,..,6
Input: 2

Human P         Player 1        pot
900$            950$            0$
BB              DEALER

Your cards:
♠A  ♠Q
Table cards:
```

```
Player 1 folds ($50/$950)

Human P        Player 1       pot
900$           950$           150$
BB             DEALER

Your cards:
♠A  ♠Q
Table cards:

Human Player wins this round!

Human P        Player 1       pot
1000$          850$           0$
DEALER         BB

Your cards:
♣4  ♦Q
Table cards:

Your status:
money: $1000 bet: $50
Please enter command (RAISE($150), CALL($50) or FOLD)!
Input: CALL
Human P calls ($100/$950)
Player 1 raises ($200/$750)
Your status:
money: $950 bet: $100
Please enter command (RAISE($300), CALL($100) or FOLD)!
Input: CALL
Human P calls ($200/$850)
Flop cards dealt:
♦A ♦7 ♥9
…
…
…
Please enter command (RAISE($150), CALL($50) or FOLD)!
Input: QUIT
Are you sure you want to quit (Y/N)?
Y
BYE BYE!!%
```

Appendix: UML



<<singleton>>
GameUI
+<<static>> getInstance(): GameUI*
+clearScreen(): void
+printWelcome(): void
+printStart(): void
+printTurn(players:std::vector<Player*>, humanPlayer:Player*,pot:size_t, cards:std::vector<Card*>,dealer:size_t, sb:size_t,bb:size_t): void
+printEnd(): void
+printExit(): void
+printInput(): void
+printInputError(): void
+printHelp(): void
+printWinner(winner:size_t): void
+print(string:std::string): void
+readInput(): void
+getLatestInput(): Command
+printAction(player:Player*): void
+...()
<<singleton>>
StateMachine
+<<static>> getInstance(): StateMachine*
+update(): void
+changeState(state:State*): void
+getCurrentState(): State*
<<singleton>>
Game
-Game(numOfPlayers:size_t=2,moneyAmount:size_t=5000, smallBlind:size_t=50,bigBlind:size_t=100)
+<<static>> getInstance(): Game*
+<<static>> startGame(numOfPlayers:size_t, moneyAmount:size_t, smallBlind:size_t,bigBlind:size_t): Game*
+getPlayerById(id:size_t): Player*
+getPlayerIds(): std::vector<size_t>
+getPlayers(): std::vector<Player*>
+getActivePlayers(): std::vector<Player*>
+removePlayer(id:size_t): bool
+getDealerId(): size_t
+updateDealer(): void
+getSmallBlind(): size_t
+getBigBlind(): size_t
+getCallCost(): size_t
+getRaiseCost(): size_t
+setHighestRaise(raise:size_t): void
+raiseBlinds(): void
+getDeck(): Deck*
+getTable(): std::vector<Card*>
+addToTable(card:Card*): void
+clearTable(): void
+setPot(pot:size_t): void
+getPot(): size_t
StateMachine manages States
States use GameUI to print information
<<singleton>>
PreFlop
+roundCounter: static int
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): PreFlop*
<<singleton>>
State
+enter(): void
+execute(): void
+exit(): void
+executeRound(): void
<<singleton>>
Flop
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): Flop*
States use information in Game
Game owns deck
<<singleton>>
Turn
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): Turn*
Deck
+Deck()
+shuffle(): void
+drawCard(): Card*
Players use GameUI to print their actions
<<singleton>>
River
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): River*
States call Player methods
Game manages Players
Deck contains Cards
<<singleton>>
End
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): End*
<<singleton>>
Win
+enter(): void
+execute(): void
+exit(): void
+<<static>> getInstance(): Win*
Player
#id: const size_t
#hand: std::vector<Card*>
#money: size_t
#bet: size_t
+Player(id:size_t,money:size_t)
+playTurn(): Command
+getId(): size_t
+isActive(): bool
+getBet(): size_t
+setBet(bet:size_t): void
+getMoney(): size_t
+giveMoney(amount:size_t): void
+getHand(): std::vector<Card*>
+setHand(card1:Card*,card2:Card*): void
#raise(): void
#call(): void
#fold(): void
Card
+color: const size_t
+value: const size_t
+Card(color:size_t,value:size_t)
+toString(): std::string
Player has Cards
Win state uses HelperTools to determine winner
HelperTools
+calculateHandValue(cards:std::vector<Card*>): std::vector<int>
+calculateOdds(rowVal:int,players:int,handRowVal:int): std::vector<double>
AiPlayer uses HelperTools to get victory probabilities
HumanPlayer
+PlayTurn: Command
AiPlayer
+PlayTurn: Command