Artificial Poker Player
IT3105

Nordmoen, Jørgen H.
Østensen, Trond

September 15, 2012

Abstract

In this text we will explain our implementation of an A.I. Poker player in the course IT3105. We will explain each phase of the project, what choices we made and our decisions that have effected the outcome in each phase. We will include some results of the different phases and wrap up with our thoughts on the implementation and possible future work.

Contents

1 Basic Structure

Since we chose to go with Java instead of Python as our programming language in this course our structure might not be as simple and straight forward because of limitations or design off Java.

We chose to utilize Java because we both recognized the advantages of having the possibility of threading our code. Python does not support true multithreading[1] and since we both have good experience with Java we chose to go with that. Of course multithreading is not everything and the speed afforded to us by Java is quite apparent compared to Python. This choice however did mean that we could not use the code that was given to us in the course which meant we had some catching up to do in the starting phases.

1.1 Poker playing

We started out by implementing the basic of any card game, the cards, the deck and power rating. The later was much inspired by the code that we had gotten from the course, but implemented as a class which could be compared to others with most of the Java interfaces[2]. We spent quite some time getting this class right and have had some strange problems from time to time, but we ended up with something that can really do its job.

Most of the work of this class is just to determine the best rank that a player can get from either an array of cards or a poker hand and some community cards and be able to compare it self with other power ratings. Looking through this code there might be some odd bits and pieces that stick out, the lazy evaluation is probably a bit of a surprise. The reason for this choice is that we quite quickly realized that when we compared power ratings in the roll-out simulation most of the time we only need the rank it self because most evaluations of *compareTo* will end then and there and doesn't need to do the costly evaluation of determining which cards should be kept and which kickers to use.

To implement the game it self we used a class called *PokerMaster* which deals with all the poker playing in our code. To enable it to support multiple phases of poker players we designed an interface which allows the *PokerMaster* to interact with all of the phases in a uniform manner. This interface, which can be found in *PokerPlayer*, supports all the methods that we needed in the later phases, but have seen several revisions before it got to this stage. This interface is then backed up by an abstract class which implements some of the tasks which every phase needed.

[1] `http://wiki.python.org/moin/GlobalInterpreterLock`
[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`

AbstractPokerPlayer does most of the work regarding chip count and make sures every phase pays what they need in order to participate further in a game. It also makes sure that blinds are payed when that is needed. When it comes to paying to the table we decided, for simplicity, that we would allow each phase to go negative. This has some ramifications for the game, because it means that there could potentially be much more chips involved than we intended at the start, but that have not been a huge problem as we force some of the phases to bet less when they are out of chips.

To make it easier for us on determining which player goes when we created the *PokerTable* class which contains some methods for keeping track of the blinds at the table and also which player is big and small blind. The *PokerMaster* uses this class to retrieve the small and big blind and also who to deal cards from.

To give each player cards we designed the *PokerHand* class which is just a helper class to enable us to compare hands, easily create power ratings and have a simple way of passing two cards around. It can compare it self to other hands, but this is not used much since we mostly deal in power ratings.

Since we share the *PokerMaster* code between the different phases there is not much else to say about this code in regard to the phases. We have some opponent modeling code which is only used by phase 3, but we will come back to that in section **??**.

2 Roll-out simulation

As the code progressed we started looking at roll-out simulation in phase 2 since that was one of the big challenges in this project.

We started by looking at what we needed to simulate a complete poker game and came to the conclusion that we should make a separate poker simulator from *PokerMaster*. Since the roll-out simulation is not dependant on any players playing the game or any notion of betting and calling we decided that the *PreFlopMaster* would only deal cards and compare them at the end of the game to speed up the calculation. The *PreFlopMaster* gets the hand we want to gather statistics about and the number of players to play against. It then deals out cards to the "other" players and begins to deal the flop, turn and river. Once this is done it declares a winner and updates its statistics for that hand. It then goes on to simulate X amount of simulations, decidable on the command-line, with the given cards and a deck to play with. Once all the simulations are done it returns a *TestResult* instance with the given wins, ties and loses and a ratio.

As we mentioned before we decided to go with Java because we realized that this roll-out simulation could easily be threaded to increase performance and so we did. We start by creating all the possible poker hands according to the hole-card equivalence classes[3]. Then we split these poker hands into lists of equal lengths with the number of lists equal to the number of processors reported by Java[4]. We then create *RolloutSimulators* which get a list each which then for each card create a new *PreFlopMaster* and simulates from 2 to 10 players and writes the results to a file. From what we can see this threading makes it

[3] `www.idi.ntnu.no/emner/it3105/lectures/ai-poker-players.pdf`

[4] `http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html#availableProcessors%28%29`

possible to do quite a lot more simulations than without the threading and the factor that dominates the time is the shuffling of the deck which is done 7 times for each simulation.

From our limited experiments this allows us to do 100 000 simulations per hand with 2 to 10 players in about 15 minutes on a 4 core Intel i7 with Hyper-Threading.

3 Hand strength

As the roll-out simulation had started to work at this point we started looking at the phase 2 player and started to implement *Hand strength* calculation. Hand strength is used after the flop in order for the phase 2 player to determine if the current hand can be beaten by other hands. This is done by comparing the current hand and the current community cards to all other combination of hands that is possible to achieve without the cards in the hand or on the table.

This to is a simulation that can be done in parallel and we did this to.

We have implemented two different calculations the one described in our handed out materials which does not take into account the possible turn and river cards and one which does. The first one is simply implemented by creating a fresh deck, remove the cards on the players hand and the given community cards and then create all permutations of poker hands that is possible to get and then compare each with the players hand to see how many times the player wins, ties and loses. To run this in parallel we just split the poker hand permutations into lists as before and calculated all the wins, ties and loses in parallel before we collected all the statistics together and return a hand strength ratio.

For the second implementation we deal out all possible turn and river cards, again except for the players cards and the community cards, and create all possible poker hand permutations and do the same calculation as above. The result of this calculation is the lowest possible score that a pair of cards can give, meaning it is much more accurate than the above calculation.

A flaw in the first hand strength calculation above is that it will most likely over estimate the strength of a hand after the flop. For instance lets say the player we are looking at are sitting with two aces on h*s hand and one ace is on the table together with a 7 and a 10 of some suits. Since there can not be a straight present and lets say for this example that only the 7 and 10 are of the same suit, while the ace on the table is of a different suit there can not be a flush either. So the simple hand strength calculation reports that the strength is 1.0. This might be a basis for a player to bet because he/she thinks that they have the best cards and nothing can beat them. But the flaw is that the turn could be a card with the same suit as the 7 and 10 and then someone might have a flush or even a straight might begin to appear. The second implementation would account for this and give the lowest ratio that could possibly appear which would be an underestimate, but it could be much more accurate because of it doesn't overestimate for the player.