Pot Limit Badugi

Ilkka Kokkarinen, ilkka.kokkarinen@gmail.com
Programming project for the course CCPS 721 Artificial Intelligence I
Chang School of Continuing Education, Ryerson University, Toronto, Canada

Rules of the game

Like all forms of poker, the card game of badugi is an excellent microworld for teaching artificial intelligence and decision theory. The game of badugi involves competitive decisions under uncertainty very much like poker, but the hand evaluation and estimation of possibilities in badugi are massively simpler than in the familiar game of poker that has far more complex combinatorial hand rankings.

The rules of badugi are well explained in the Wikipedia page. You can also watch this Youtube video to hear one player explain and comment on the game mechanics and tactics while he plays, although note that this video is about **limit badugi** where the bet sizes for each round are fixed. Our project modifies the rules of Badugi slightly in the following fashion:

1. There are no blinds: instead, both players make an equal forced **ante of one chip** before receiving their cards.
2. The sizes of bets and raises are **pot limit**, the size of this limit at the moment depending on the betting round and the current number of chips in the pot. (The framework will always tell your agent how much it is allowed to bet and raise, so you don't have to worry about the exact rules that govern this.)
3. At most four raises are allowed per betting round, including the first bet. Attempting to raise the fifth time is merely considered a call that ends that betting round. (This rule is necessary to prevent two runaway agents from raising each other forever and that way crash the system.)
4. To avoid the complexities of short stacks and all-in situations, both players are assumed to have sufficiently deep stacks that will never run out of chips during a hand. These chip stacks get silently replenished between each hand. Each hand is therefore a separate episode for the agent, in theory independent of the other hands before and after it.
5. One full **heads-up match** between two agents consists of **100,000 individual hands**, to minimize the effect of luck compared to the effect of skill in the long run between agents that have an actual skill difference between them. The agent with the higher total count of chips won over these hands **receives two matchpoints in the tournament**. The magnitude of the chip count does not matter, only the sign of the difference between the final chip counts of the players, so winning the match by one chip gives the winner the same two matchpoints as winning that match by 100,000 chips. If there is a tie (rather unlikely, but theoretically possible), both agents receive one matchpoint.

6. The **tournament** consists of a series of heads-up matches so that every agent plays one full heads-up match against every other agent. The results of the tournament are tallied by ranking the agents by the number of matchpoints that they collected from their individual matches.

The Badugi Java Framework

The instructor provides the following Java class framework that you embed your Badugi playing agents in. All agents must implement the `PLBadugiPlayer` interface so that they can all seamlessly play together through the `PLBadugiRunner` tournament framework.

The following classes should not be edited by the students in any way, except for changing the number of rounds per match and similar parameters, and the participating agent class names in both the `main` and `playThreeHandTournament` methods of `PLBadugiRunner`. If you think that you have found a bug or want to suggest some kind of real improvement, email the instructor about it. The first finder of any bug or suggestion of a good improvement to the code receives a reward between 1 to 3 marks added to his or her course grade, depending on the severity of the discovered bug or the quality of the suggested improvements.

Class	Description
`Card`	Individual playing cards that comprise the deck.
`EfficientDeck`	A utility class used by the framework to simulate a deck of cards and its operations. (Back in the first version of this course in 2016, I used a far less efficient class to represent a deck of cards and its operations, hence the name. But this one makes all operations lazy and work in guaranteed O(1) time.)
`PLBadugiPlayer`	Interface that defines the methods that all badugi agents must implement.
`PLBadugiHand`	A utility class to represent a four-card badugi hand and its operations. Saves some work for the students so that they can concentrate on the AI aspects of this programming project.
`PLBadugiRunner`	The methods to play one hand, a heads-up match, or an entire tournament.

Rules for the agent code

Each agent subclass must have **a default constructor** so that it is possible for the Badugi framework to create an object of your class using reflection. The one and the same agent object will be used throughout the entire tournament to play against all other agents. Therefore, you can have your agent adjust its

internal parameters to learn from its experiences during the tournament, since the modelling of opponent tendencies is important in the real world where we are all flawed and nobody plays perfect Nash equilibrium poker but somehow deviates from the optimal strategy. For example, every time your bluff works, the agent might increase its future aggression and bluffing probabilities. Conversely, whenever your bluff is caught, decrease these parameters. Bet and raise more against opponents who tend to be passive, and tend to believe their bets and raises more than those of a wild opponent who is betting and raising almost all the time, making it statistically impossible for him to have a good hand every time he does that.

The methods of your agent must not use excessive memory or time. All decisions should be made in a split second so that we will be able to run the complete heads-up tournament as a single batch run taking at most an hour or two in total. Make sure to test the final version of your agent over the play of millions of hands before submitting it, to ensure that it will not crash or get stuck in an infinite loop in any situation. (Even if you are working alone, you can always run the tournament with multiple instances of the same agent class.)

The agent must be silent at all times, **and not emit any console output whatsoever during its execution.** Before submission, please make sure to comment out (or somehow otherwise turn off) all debugging printouts that you used during the development stage. This should again be part of your private test run where you make your agent play millions of hands.

The agent must play honestly within the rules of badugi so that it may not not use **reflection** or any other underhanded means to change its cards or peek into the internal details of the framework or the other agents in the tournament. **Also, exploiting any bugs or security holes in the badugi framework is considered academic dishonesty and will automatically result in a zero mark for the project.** Please report any such bugs and security holes that you may find to the instructor rather than exploiting them so that you receive a proper reward for your discovery in the form of a small bonus to your course grade, instead of receiving a disciplinary notice in your Ryerson academic record. (It really should not be difficult to choose the right action here.)

Your agent is not allowed to write any data files during its execution. However, the agent is allowed to **read data from a file once during the time when it is being constructed**. This data file, if you have prepared one for your agent, must be submitted alongside the agent class, and named something that contains your student number to ensure that two agent submissions don't accidentally try to use data files with the same name.

You are allowed to freely discuss badugi tactics and theoretical ideas of AI programming with each other, but not give each other any concrete Java source code. However, so that you can properly test and debug your agents and get an idea where your agents are currently standing, **you are allowed to pass around your agent class files in compiled bytecode form** (any sort of decompiling back to source code is forbidden) and this way organize mini-tournaments among study groups of your fellow students. The source code of the submitted agents is inspected by both eye and automated plagiarism detection tools,

and if copied code is discovered, both the plagiarist and the original author will receive a penalty. Don't give your class bytecode to anybody who you don't fully trust to be honest here.

Even if you are working alone, a good approach might be to start with a simple agent that bets and draws cards according to some simple heuristic rules. Then create a new more advanced version that is designed to beat your current version because of its weaknesses and deficiencies. Once the new version handily beats the current version, this new version becomes your current version, and you start working on the next version that is designed to beat your new current champion. Iterate this process as far as you can within the constraints of your time and energy.