EP White

Introduction

EP White’s bot is a no frills CFR bot implemented in Python.

Poker engine and framework

The engine provided by the pokerbots team wasn’t adapted to the approach we wanted to take. We implemented a River of Blood engine following an abstraction specifically developed to plug into our equilibrium finding algorithm. Some principles we followed:

- Games implement a protocol designed to be convenient for equilibrium finding algorithms
- Game states are immutable objects
- Chance is not treated as a player, instead game states (game tree nodes) have a chance attribute
- Chance states need not give access to the full list of chance actions and my instead implement a sample method: this is very useful for Monte Carlo algorithms
- Games do not directly give access to the list of possible actions for the active player, but instead implement an infoset method, which returns another protocol
- The InfoSet protocol gives access to an actions method, which yields the authorized actions at this infoset

This simple set of abstractions allowed the development of a very flexible game abstraction framework. One can impose a monoidal structure on the set of abstractions of a game. Letting A and B be different abstractions, A * B is the smallest abstraction which allows to differentiate between game node trees whenever either A or B allows it. The identity element is obvious. We call this idea algebraic abstractions. In practice we develop algebraic abstractions as objects which give access to a subset of the information available in an infoset. For example, a pot abstraction could round the pot value to the nearest power of 2.

Betting abstraction

Abstraction

After playing around with various betting abstractions, it became clear that the simplest one we tried was the strongest. Two hypotheses which could explain this:

- We overfit to the abstraction
- We never converged to equilibrium due to the more complex structure of larger betting abstractions

Our final betting abstraction contains the following actions:

- Fold
- Call
- Check
- RaiseHalfPot
- RaisePot
- Allin

The actions RaiseHalfPot, RaisePot and Allin can correspond to different actions for different game tree nodes, even at the same infoset. We refer to them as pseudo actions. We disallow more than 8 consecutive bets on the same betting round. Although 8 seems large, this seemed to make the abstraction more robust. This also avoids calling whenever the opponent 4-bets all in. Eliminating this kind of variance is key to performing well because of the check fold strategy.[1]

Action translation

The opponent unfortunately (or fortunately ?) does not share our betting abstraction. We map their actions to legal actions within our abstraction using a similarity metric. Bets and raises are compared relatively using the formula min(a / b, b / a). Allin is a little special as it forbids the next player from raising. See the Translation class in the abstraction to see how this is handled.

State abstraction

Cards

The card abstraction is based on the equity distribution. For each street, to each hand and community, we associate a hand strength histogram by repetitively dealing the rest of the community (monte carlo) and evaluating the hand strength. The distributions thus obtained are clustered using KMeans and the EMD metric, which is trivial to implement in the 1D case.

We tried a few variants on this idea which turned out weaker:

- Potential distribution: don't deal the full community but only the next street, and evaluate made hands then ; the idea is that the bot shouldn't assume they won't fold until showdown: there is such a thing as equity realization and hand playability
- Opponent Clustering Hand Strength: same idea, but evaluate hand strength against various clusters of opponent hands. This is much slower and we didn't implement a fast enough version to fully converge to equilibrium

Our best bot uses 8 clusters on the preflop and 40 clusters for all other streets (until the 9th card, or 4th run). Distributions are 50 dimensional vectors. We couldn't go with more clusters because Python is no C++ and we realized C++ would be beneficial too late to confidently implement it. The card abstraction was our bottleneck and it seems likely that a C++ implementation of our ideas would have been first by a large margin. It remains to be seen whether our bot will approach first place nonetheless.

Monte Carlo was heavily used and the number of iterations became an important parameter. Upon visual inspection, fewer iterations gave less accurate infoset retrieval, but the performance cost of accuracy was too high to justify significantly more iterations.

Other abstraction ideas

Our abstracted information sets contain the following information (they are the product of the following algebraic abstractions):

- The pot, which was abstracted into 20 buckets of equal size ; experimentation suggests this is better than abstracting the pot odds (cost / pot + cost)
- The street number
- Whether the last card is black (will there be a run ?)
- Which player we are
- Which actions we have access to (the betting abstraction is also a state abstraction, since it is a function of the abstracted infoset)
- Card abstractions as described above

[1] The check fold strategy is an abomination and can be dealt with easily: simply deal random amounts of hands 🙂.

Equilibrium finding

We implemented a few variants of CFR:

- CFR
- CFR+ (although it isn't quite what the paper describes)
- External Sampling MCCFR
- External Sampling MCCFR with linear discounting

The latter version was used for essentially all experiments once it was deemed superior. All the implementations handle the Game and InfoSet protocols and were tested successfully against other imperfect information games:

- Rock Paper Scissors
- Kuhn Poker
- Nash Bargaining

We needn't ever implement game abstractions as separate games because MCCFR need only be able to sample at chance nodes, rather than exploring all chance actions. Game abstractions were subclasses implementing the infoset method. The simplicity of this scheme speaks for itself. Here is an example abstraction implementation:

```
from dataclasses import dataclass

from cfrpy.abstraction import abstract
from cfrpy.pkr.abstraction.abstractions import Translation, capped_bets
from cfrpy.pkr.game import RiverOfBlood

from .basic import run, street, player, linearpot, basic, equities


actions = capped_bets(basic, 8)
cards = equities((8,) + (40,) * 7, dimension=50, maxiter=30)
states = run * street * player * linearpot(20) * cards * actions


@abstract(Translation, states, actions)
@dataclass(frozen=True)
class Abstraction(Translation):
    ...


@abstract(RiverOfBlood, states, actions)
@dataclass(frozen=True)
class TrainingAbstraction(RiverOfBlood):
    ...
```

The implementation was Python and performance was an issue. The card abstractions were the bottleneck and were largely implemented in Cython by extending eval7. This was not enough to scale to abstractions larger than a few tens of thousands of information sets.

Conclusion

CFR seems like a very naive algorithm compared to the state of the art in artificial intelligence. It is essentially a collection of regret learners, one for each infoset, connected only by transition probabilities. Game trees are rich with structure that CFR seems not to take advantage of. Yet it is the state of the art in imperfect information games.