No Limit Texas Hold'em Versus a Shover

Bryan Coutts

August 14, 2020

1 The Problem

Poker has always been a fascinating game to me, as on the surface it seems to be a game of chance, but there turns out to be a great depth of skill in probabilistically playing well and gaining an edge over one's opponents, winning more hands in the long-term. I'm interested in the balance is between luck and skill in poker, and it's natural to investigate a simple special case of the game. Specifically, I want to investigate the scenario where you're playing:

- No Limit Texas Hold'em (NLHE) poker
- Heads-up (versus only one other player)
- With blinds of \$1 and \$2, each player starting with \$100
- Where the opponent always goes all-in at the beginning of each hand

Naturally, choosing to go all-in every turn without even considering your cards requires no skill. Thus, investigating the win-rate such a player would have, versus a player who is playing optimally to counter their strategy, would establish a lower bound on the role of luck in heads-up NLHE poker. We wish to put ourselves in the shoes of the optimal player, and determine what the optimal strategy is.

Unsurprisingly, this problem has been considered before; in Will Tipton's *Expert Heads Up No Limit Hold'em, Vol 2*, Section 16.2.1, the optimal winrate is computed to be roughly 64%. The technique used is a common technique for finding optimal strategies in game theory; one starts with some arbitrary strategy, finds ways in which it is suboptimal, and keeps iteratively improving it until it converges to an optimal strategy. Here, I'll be outlining an interesting alternative approach using linear programming. In addition to being an interesting use case for LPs, this also provides a bit more certainty; iterative approaches like the one used above always make me a little nervous, as they're susceptible to getting stuck in local optima (although I believe in this particular case it does work).

2 The basics

Before getting into the mathematics of solving this ourselves, we can make some elementary observations. The natural way we can gain an advantage, is by waiting for good hands, which have a good chance of beating our opponent's uniformly random hand. In each round, we know our opponent

will go all-in, and thus (having already put in our chips for the blind), we have only the choice of whether to go all-in, or to fold. This strategy may also depend on the chips each player has (e.g., as our stack dwindles, they may be forced to settle for hands with lower winrates).

We also note that this is effectively a 1-player game. The opponent's strategy being fixed, we simply have a series of rounds in which we see our 2 cards and decide whether to call or fold. We can even precompute the winrate our hand will have vs. our opponent's uniformly random hand from the remainder of the cards. From this, a tempting line of reasoning would be to have some kind of recursive solution to this problem: if when making this decision we can determine the expected winrate of folding, vs. the expected winrate of calling, then we should make the move that results in a higher winrate.

For simplicity, let's ignore the blinds for now, and suppose that each player just puts in \$1 each round before seeing their cards, and let $x_{k,h}$ represent the optimal winrate when you have k chips and your hand is h. If we have k chips, the outcomes change a bit based on whether $k \leq 100$. If $k \leq 100$, calling and losing means we lose the game, and winning means we double up to $2k$. If $k > 100$, calling and winning means we win the game, and losing works out to leaving us with $2k - 200$ chips. This leads to the following recurrence:

$$x_{k,h} = \max\Big[\underbrace{x_{k-1,*}}_{\text{fold winrate}}, \underbrace{w_h x_{2k,*} + (1 - w_h) \cdot 0}_{\text{call winrate}}\Big] \qquad \text{for } k \leq 100$$

$$x_{k,h} = \max\Big[\underbrace{x_{k-1,*}}_{\text{fold winrate}}, \underbrace{w_h \cdot 1 + (1 - w_h) x_{2k-200,*}}_{\text{call winrate}}\Big] \qquad \text{for } k > 100$$

where w_h is the winrate of the hand h versus a random opponent hand, and $x_{k,*}$ is a shorthand representing the winrate when we have k chips and a randomly-chosen hand:

$$x_{k,*} = \sum_{\text{hands } h} [\text{chance of getting hand } h] \cdot x_{k,h}$$

Our main problem at this point is that this recurrence is cyclic. If we were to try to compute, say, $x_{100,\text{A9o}}$ (our winrate at the start if our first hand is Ace-9 offsuit) using this reccurence, we'd need to know the value of $x_{99,h}$ for each h, which in turn would require us to compute $x_{98,h}$ and $x_{198,h}$. Eventually, we would reach $x_{50,h}$, which would then contain $x_{100,h}$ in its recurrence, giving us an infinite loop.

3 Linear programming to the rescue!

This is where linear programming comes in. If you're unfamiliar with linear programming, the basic gist of it is that a linear program (LP) is a type of problem where you minimize (or maximize) a linear function, subject to linear constraints. For instance, an LP might look like this:

$$\begin{aligned} \min \quad & 3x + 4y - z \\ \text{subject to} \quad & 2x = 4z + y \\ & 3y + x \leq z \end{aligned}$$

These are very similar to linear systems of equalities that one might be familiar with from linear algebra, except we are allowed to use $\leq$ and $\geq$ inequalities, in addition to equalities. A problem of this form is called a linear program (LP); the goal is to find values x, y, z which minimize the *objective function* $3x + 4y - z$, while obeying the constraints. LPs are perhaps the most fundamental class of problems in the field of mathematical optimization, and there exist fast solvers that can take an arbitrary LP and produce its optimal solution; many real-world problems are solved in this way. I won't go over how LPs are solved here, but the takeaway is that if we can simply model our problem as a LP, we can use an existing solver to solve it.

Back to our problem. Ideally, we would like to write an LP whose constraints just require our recurrence relation to hold; then, any solution (actually, there would only be one) would solve our problem. However, max is not a linear function. Fortunately, there's a trick to working with max in linear programs: if we want to compute $c = \max(a, b)$, we need $c \geq a$ and $c \geq b$, but also want to minimize c. This can be encoded in the LP:

$$\begin{aligned} \min \quad & c \\ \text{subject to} \quad & c \geq a \\ & c \geq b \end{aligned}$$

So for our problem, we have a bunch of constraints where a winrate $x_{k,h}$ must be the max of the winrate from calling and folding. We can use the above trick to encode these constraints as an LP, where we minimize every winrate $x_{k,h}$, subject to $x_{k,h}$ being at least the fold winrate (the fold constraints), and at least the call winrate (the call constraints). This culminates in the following LP:

$$\begin{aligned} (P) \qquad \min \quad & \sum_{k=0}^{200} \sum_{\text{hands } h} x_{k,h} \\ \text{subject to} \quad & x_{k,h} \geq x_{k-1,*} && \forall k, h \\ & x_{k,h} \geq w_h x_{2k,*} + (1 - w_h) x_{0,*} && \forall h, k \leq 100 \\ & x_{k,h} \geq w_h x_{200,*} + (1 - w_h) x_{2k-200,*} && \forall h, k > 100 \\ & x_{k,*} = \sum_{\text{hands } h} [\text{chance of getting hand } h] \cdot x_{k,h} && \forall k \\ & x_{0,h} = 0 && \forall h \\ & x_{200,h} = 1 && \forall h \end{aligned}$$

In the above LP, the first constraint line encodes the fold constraints, and the next two lines encode the call constraints. The 4th line enforces that $x_{k,*}$ represents having k chips and a random hand, and the last 2 lines encode that having \$0 is a loss, and \$200 is a win. And with this, we're mostly done. There are a few minor details to worry about when actually constructing this LP; feel free to skip the next paragraph if you aren't interested in the gritty details of constructing this LP.

I've been ignoring blinds to make this explanation simpler, but when actually constructing this LP we need to take this into account, by counting being in the small blind or big blind as different states; as such, we have a winrate $x_{k,b,h}$, for every chip amount k, blind position b (either small or big blind), and hand h. In addition, although there are 2652 unique 2-card hands, many of these (e.g. 9 and 10 of hearts vs. 9 and 10 of spades) are effectively the same. Ultimately, there are really only 169 distinct 2-card hands up to equivalence (13 pairs, 78 off-suit hands, 78 suited hands);

this lets us cut down drastically on the number of states/variables. Finally, there's an arbitrary decision of whether to make $x_{k,*}$ an explicit variable, or just write it as a linear combination of other variables in the other constraints; it turns out the former makes the problem much easier for GLPK to solve, so I've done that. When all is said and done, we have an LP with 68341 variables and 134927 constraints; a somewhat large but tractable LP. On my computer, GLPK is able to solve this LP in about 30 minutes.

And when we run our LP solver, we finally have our answer! When playing optimally against an opponent that always goes all-in, you can expect to win $x_{100,*} \approx 64\%$ of your games. The solution to the LP can also be used to actually play optimally, by using it to compute the expected winrate of calling or folding from any position, and determining which is higher.

4 Extra Stuff

Here I'll talk about some more advanced stuff that builds on this, with less detailed exposition.

4.1 More general stochastic games

The technique we used here is really quite general. It didn't matter much that our setting was poker; we simply needed a game in which you have choices of ways to move probabilistically to new state(s), so that for each move we introduce an inequality that the winrate of a given state, is at least the winrate obtained by performing the given move. This will not only allow us to consider a broader class of problems, but also let us consider our LP and solution more abstractly and with fewer tiny details; something I greatly appreciate as someone who likes to pretend to be a pure mathematician!

Let's say a stochastic game is a set of states Σ, with some subsets W and L of terminal win and loss states. For each state $\sigma \in \Sigma$, we have a set of moves M_σ, where each move $m \in M_\sigma$ moves probabilistically to a new state in Σ, and the probability of moving to the state $\sigma' \in \Sigma$ is denoted $\Pr[m \to \sigma']$. This would result in the LP:

$$
\begin{aligned}
(GP) \qquad \min \quad & \sum_{\sigma \in \Sigma} x_\sigma & & \\
\text{subject to} \quad & x_\sigma \geq \sum_{\sigma' \in \Sigma} \Pr[m \to \sigma'] x_{\sigma'} & & \forall \sigma \in \Sigma, m \in M_\sigma \\
& x_w = 1 & & \forall w \in W \\
& x_\ell = 0 & & \forall \ell \in L
\end{aligned}
$$

One might naturally call such a game a *stochastic game* (although in the literature, this term has many conflicting definitions, none of which seem to agree with my use case). The above could then be considered a general solver for 1-player stochastic games, and indeed, I've implemented it as a general solver `solver.py`, with a client `poker.py` that uses it to solve the poker-related problem. I havent been able to come up with other interesting and motivating examples of these games for my solver to solve; such an example would both need to be interesting, and ideally involve a cyclic recurrence so that it can't be solved via recursion and dynamic programming, thus necessitating the LP approach. If anyone can think of any, please let me know!

4.2 The dual problem

It's typical when modelling problems as LPs, to take the dual LP and interpret it in some way. For those unfamiliar with LP duality, the gist of it is that for any minimization LP (P), there's a straightforward way to construct a maximization LP (D) called its *dual*, such that (P) and (D) have the same optimal value, and every feasible solution in (D) has lower objective value than every feasible solution in (P). Every variable in (P) corresponds to a constraint in (D), and every constraint in (P) corresponds to a variable in (D). The exact meaning of the dual problem is, in general, pretty mysterious and difficult to interpret for an arbitrary LP, but often has a nice interpretation in a specific context (see, for instance, the example of min-cut and max flow).

If we look at the problem (GP) (from this section) and take its dual, we obtain, with some massaging:

$$
\begin{aligned}
(GD) \qquad \max \quad & \sum_{\sigma \in \Sigma} \sum_{m \in M_\sigma} \sum_{w \in W} \Pr[m \to w] \cdot y_{\sigma,m} & \\
\text{subject to} \quad & \sum_{m \in M_\sigma} y_{\sigma,m} = 1 + \sum_{\sigma' \in \Sigma} \sum_{m \in M'_\sigma} \Pr[m \to \sigma] \cdot y_{\sigma',m} & \forall \sigma \in \Sigma \\
& y_{\sigma,m} \geq 0 & \forall \sigma \in \Sigma, m \in M_\sigma
\end{aligned}
$$

This can be interpreted as a sort of network flow problem, where the flow represents the probabalistic movement of our strategy from state to state. We have a net flow of 1 into each game state, and each variable $y_{\sigma,m}$ determines how much of the flow into σ gets sent out through the move m. In fact, if there's a unique optimal move m for state σ, m will be the only move in M_σ to receive a nonzero value for $y_{\sigma,m}$.

This is interesting for two reasons. For one, this is the easiest way to read optimal moves from the LP solution. If you're in state σ, instead of explicitly computing the winrates of each move $m \in M_\sigma$ to take the best one, you can simply choose the m for which the dual variable $y_{m,\sigma}$ is greatest (and most LP solvers will give you this dual solution as well). Secondly, this dual LP represents an equally valid approach to solving the problem; we could instead have come up with this network-flow approach and created this LP as our solution to the problem, and it would have worked just as well.

4.3 Minimizing turns taken

In some games, rather than maximizing winrate, one might want to minimize the number of turns taken (e.g. a maze). Our approach can also solve these problems with some small modifications. Our primal variables x_σ will now correspond to the expected number of moves to finish the game starting from x_σ, and the terminal (winning) states have value 0 instead of 1. Recursively, we require that x_σ is at most 1 greater than the expected number of turns from each of its possible moves, and maximize each x_σ instead of minimizing. This is included in the code by setting the `gs.min_turn_mode = True` flag on the GameSolver object.

One example of a new problem this lets us solve is the egg drop problem (solved in `egg_drop.py`), a well-known tech interview question in which one has a building with N floors, and has E eggs, and must determine the highest floor one can drop an egg from without it breaking. This can be thought

of as a stochastic game, where a move amounts to deciding which floor to drop the next egg from, and then you move probabilistically to a new state, depending on whether the egg broke or didn't break. Admittedly, this isn't the greatest showcase of my solver; since the underlying recurrence has no cycles, one could easily just solve it via dynamic programming rather than bother using an LP. Still, it's nice that there are interesting problems that can be solved by just describing them to this solver.

4.4 2-player stochastic games

It'd be nice if we could solve 2-player stochastic games with this approach (one particular example I considered was tic-tac-toe, except you have some probability p of failing to place your symbol). In fact, it's almost possible to use our approach by taking each state where it's our opponent's turn, and flip its inequalities so its winrate is **at most** the winrate from each move they can make, and maximize this winrate instead of minimizing. Unfortunately, this doesn't quite work, as the minimization and maximization of different winrates can be at odds. For instance, consider a simple game where there are 10 our-turn states $\sigma_1, \ldots, \sigma_{10}$, which each have one move leading to an opponent-turn state σ_{11}, which in turn leads to a single win state σ_{12}. The true winrates would be $x_k = 1$ for each k, but the optimal solution to this LP would be $x_k = 0$ for each k.

I've considered a couple solutions to this problem. I've noticed some simple problems do appear to work in spite of the above issue; it's possible we could find some class of stochastic games for which this approach works. Perhaps if moves are "symmetric" in some way, we avoid problems like the above where the our-move variables force the opponent-move variables down or vice-versa.

Another approach would be similar to the branch-and-cut approach to solving integer programs, where we solve the above LP, and when we find cases where a state like σ_{11} is "loose", i.e. not equal to the winrate of at least one of its moves. For each of these cases, we could then add a constraint to force x_σ to be at most the greatest of its current winrates (or if it's an opponent's state, force x_σ to be at least the least of its current winrates), and re-solve the LP. However, it's possible that if the winrates of its moves were also incorrect, that x_σ variable would still be loose in the new LP, requiring us to keep iterating until we remove all loose variables, and it's unclear how many iterations of this would be necessary.