Pokerbots Course Notes

Massachusetts Institute of Technology
6.176: Pokerbots Competition*

IAP 2021

Contents

*Contact: pokerbots@mit.edu

1 Lecture 1: Introduction to Pokerbots

This lecture is taught by Stephen Otremba.[1] All code from lecture can be found at the public github.com repository mitpokerbots/reference-lecture-1-2021. The slides from this lecture are available for download on Canvas and at the public github.com repository mitpokerbots/class-resources-2021.

At this lecture (and all others), we raffle off a pair of Sony Headphones. Synchronously attend lectures if you want a chance to win them!

We'd also like to thank our 10 Pokerbots 2021 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at pkr.bot/drop to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

1.1 Class Overview & Logistical Details

There will be six 90 minute lectures running on MWF 1:00 – 2:30pm EST from 1/4 to 1/15 held at pkr.bot/class, and office hours will be held during the first three weeks of IAP at pkr.bot/oh. There will also be a live scrimmage server for the first three weeks, on which you can challenge any other team in the class as well as our reference bots. Weekly tournaments will be held on the scrimmage server every Friday night, and there will be prizes for winning teams. The final tournament and event will be held on January 29, 2021, which is where the Pokerbots 2021 winners will be announced. The final event will also feature more prizes, an expert guest talk, winning strategy analysis, a chance to play against the bots, networking with sponsors, and more! This year's Pokerbots prize pool is over $30,000, distributed over many different categories—the syllabus lists many of the categories we will be awarding. The six lecture topics will be as follows:

1. Introduction to Pokerbots
2. Poker Theory
3. Game Theory
4. Engineering and Performance
5. Advanced Topics
6. Guest Lecture: Noam Brown, PhD[2]

To receive credit for the class, you must submit bots to the scrimmage server. Your bot for each week has to defeat your bot from the previous week, as well as a skeleton bot in a one-shot tournament; otherwise, you will be asked to submit a one-paragraph makeup report describing strategies you tried over the week. At the end, you must also submit a 3-5 page long strategy report.[3] More guidelines will be announced later in the course.[4] Take special care to read the Rules and Code of Conduct on GitHub and Canvas.

1.2 Introduction to Poker

For this section, we will be talking about the game known as heads-up no-limit Texas Hold'em ("poker"). The objective of poker is to win as many chips as possible. Players bet into a pot in several rounds, and the pot is won by the player with the better poker hand at the end. In a single betting round, the first player can either bet 0 (check) or any amount between the "big blind" and number of chips they have left. If they check, action passes to the second player, and if they bet, the second player can *fold* (quit the round), *call* (bet the same amount as the first player), or *raise* (bet more than the first player, up to the number of chips

[1] Email: sotremba@mit.edu.

[2] Noam created Libratus and Pluribus, two of the most successful and famous pokerbots!

[3] You are welcome to include images or code snippets in your final report, as well as discuss strategies you attempted that did not pan out. This is an open ended report, and is for us to gain insight about how you approached the Pokerbots challenge.

[4] All class details are included in the syllabus, available on GitHub and Canvas, with more details coming soon.

they have left). A player's final "poker hand" is determined as the best five-card hand that can be formed out of seven cards: their unique two *hole* cards and five shared *community* cards.

The first betting round is special because it begins with blinds, a forced amount players have to bet. The next time around, there is no minimum amount.

The structure of a game is as follows: the game begins with each player receiving two hole cards. The first betting round takes place, and then the "flop" (three community cards are revealed). After another betting round, there is the "turn" (a fourth community card is revealed). Another betting round occurs, and there is the river (a final fifth community card is revealed). The last betting round takes place next, followed by settlement (cards are revealed and the pot given to the winning player).

The possible poker hands are displayed in the slides, in order of best to worst hands starting with the top left hand. The final hand is called "high card," which is none of the displayed ones. Even within each hand, there are tiebreakers if both players have the same hand. Make sure to look up, using one of the provided resources, which hands are better when building your bot.[5]

1.2.1 2021 Variant: "Blotto Hold'em"

This year, our variant of poker is called Blotto Hold'em.[6] Blotto Hold'em is based on the popular poker variant Texas Hold'em with a modification that you will be playing three distinct boards of poker, much like in Blotto.[7] At the beginning of the game. six cards are dealt to each player; bots must then *assign* these cards into three pairs of hole cards.[8] Each board is also pre-inflated with one, two, and three big blinds respectively, so the pots do not start empty. After assignment, the Pokerbots play standard no-limit hold'em on each of these boards simultaneously. Much like in Blotto, your stack is shared across all three boards; you must allocate your resources across the three boards during the game. The goal is to assign your six given cards effectively and beat your opponent in heads-up no-limit hold'em on each board.

Betting in Pokerbots is also different from Texas Hold'em in that players have the same bankroll during every round (200 for this variant). Winners are calculated in terms of their change in bankroll aggregated across rounds.

1.3 Skeleton Bot Setup

Before reading the rest of this section, we recommend following the instructions in Piazza Post @10.

1.3.1 GitHub and Version Control

GitHub is a version control and code management system. Using it, *clone* the public Pokerbots repository mitpokerbots/engine-2021 to get started. If you've successfully set up Git on your machine, this can be done by navigating to the directory where you'd like to keep the code and running the command

```
$ git clone https://github.com/mitpokerbots/engine-2021.git
```

Skeleton bots for all supported languages are included in this repo.

We recommend that you create a new private repository of your own to code in on GitHub, and then set this up by cloning it into your working directory and copy-pasting the engine files and folders (`engine.py`, `config.py`, `cpp_skeleton/`, `python_skeleton/`, and `java_skeleton/`) into the clone of your own repository. On the Pokerbots GitHub, the folder "`python-skeleton/`" contains the Python 3.7 skeleton bot. There are also Java and C++ skeletons available on our GitHub repository.

To upload code from your machine, you have to create a *commit*. To make a commit, *add* the changed files you want to push, describe it with a commit message, create the *commit*, and *push* the commit online. Your partners can now *pull* your changes to their own desktops.[9] For example, after editing `player.py`, you would push it to GitHub with the following commands:

[5] The following resources are great for learning more about Texas Hold'em: pkr.bot/poker-rules and pkr.bot/poker-video.
[6] A full variant writeup is available on GitHub and Canvas.
[7] The bot plays all three boards in parallel over the course of each round.
[8] A starting hand consists of two hole cards, which belong solely to the player and remain hidden from the other players.
[9] You can learn more about this workflow at `http://web.mit.edu/6.031/www/fa20/getting-started/#git`.

```
$ git add player.py
$ git commit –m ``made our bot super cool''
$ git push
```

Table 1 lists some common Git commands for your convenience.

Command	Description
git clone **your_link**	Downloads code from remote
git status	Print the current status of your repo
git pull	Pulls latest changes
git add **your_files**	Stages changes for commit
git commit -m "**your_message**"	Commits added changes
git push	Pushes your changes to remote

Table 1: Important Git Commands

1.3.2 Connecting to the Scrimmage Server

To use the scrimmage server, go to pkr.bot/scrimmage. There, you can create or join a team with your one to three partners. To upload a bot, go to the "Manage Team" tab. Bots must be submitted as a `.zip` file, which you can easily do by going to "Clone or download" on your online GitHub repository and downloading your repo as a zipped folder. After you set one of your uploaded bots as your main bot, you can challenge any of the teams on the scrimmage server. If a team has a higher ELO rating than you, your challenge will be automatically accepted—otherwise, they must accept your challenge request.

1.4 Testing Your Bot Locally

To test your bot locally (without using the scrimmage server), you have to download the engine—again using GitHub. The engine consists of two files: `engine.py`, which runs your bot, and `config.py`, which contains parameters for your bot. The default parameters for the game (`NUM_BOARDS`, `BIG_BLIND`, and `STARTING_STACK`) are the values we will be using, so don't change those. You should feel free to change `NUM_ROUNDS` and the time-related parameters; however, `STARTING_GAME_CLOCK` is capped at 30 for our tournament, since we do not want bots pondering for a long time. Before running the engine, you must specify which bots you wish to use in your local game. This is done by providing the file path of each bot in `config.py`. For player 1 the path is `PLAYER_1_PATH` and for player 2 the path is `PLAYER_2_PATH` (you may use the same file path to pit a bot against itself). To run the engine, we will be using command line. Change the working directory as needed to your engine folder, and then run `python3 engine.py`.[10] You will be greeted by the MIT Pokerbots logo, and your game log(s) will be output.

Looking at the game logs, we can see every action taken in each game and the different boards. The log shows the actions chosen by each bot. You can also see that the log notes the flop, turn and river. The engine does not tell you what type of poker hand each player has or who won each board—this has to be determined by yourself—but it will do the calculations and tell you who won.

In addition to the game logs, you will get a dump file, and this will be done for each bot. If you put any print statements in your bot, they will show up in the dump file. If you have an error in your code, the error descriptions will be in your dump file as well.[11]

1.5 Overview of Skeleton Bot Architecture

Now, we'll look at the Python skeleton bot itself (`player.py`). The function `__init__` simply initializes the player object when a new game starts, and is useful for initializing variables that you want to access over the course of the game. The function `handle_new_round` is called at the start of every round, and the parameter `game_state` contains some information about the current state of your game. The function

[10] Depending on your setup, the command used may vary. Please refer to the setup Piazza post.
[11] The default names for these dump files are `A.txt` and `B.txt`.

`handle_round_over` is called whenever a round ends, and is thus great for updating variables using the information from the last round played.

The `get_action` function determines your bot's move based on the presented information in the function's parameters—it's where the magic happens. Each of the commented-out lines contains important variables and their respective explanations, which you will likely find very useful as you develop your pokerbot.

1.6 Coding Lecture 1 Reference Bot

We will now be implementing a basic all-in pair-hunting bot and submitting it to the scrimmage server. The first thing we need is a way to keep track of our allocations for each board being played–we will do so using a list of lists.

Next we will want to make an `allocate_cards` method to help us loop through the cards and find possible pairs. When looping through our cards, we want to keep track of the ranks of all cards we encounter. To do so, we use a dictionary that maps a possible rank (the key) to the cards of that rank (the value). Hence, we initialize an empty dictionary called `ranks`, where the keys are strings and the values are lists storing the cards matching a given rank. As we loop through our cards, if we have already encountered the card's rank we can just append our current card to the list holding the cards of that particular rank; if not, then we will want to add a new list to our dictionary with our current card's rank. Once we have looped through all of our cards then we will want to go through our dictionary ranks to identify pairs. So, if the length of the cards at a specific rank is 1, we will add the card rank to our singles list (cards that can't be involved in a pair); if the length of the cards is 2 or 4 then we have 1 or 2 pairs, and if the length is 3 then we have one pair and one single card so we will add the first two cards to our pairs list and the third card to our singles list. In the final portion of this method, we then allocate the pairs to boards by looping through the number of boards and setting `self.board_allocations[i]` equal to the cards that we want for each board. In future lectures we will go over how to optimize placement of our hole cards.

Next, we go to the `get_actions` method, because currently we have only implemented check and call logic, and we may want to wager more for a board if we know we have a pair at that board. Initializing the variable `strong_hole` in our `__init__` function will help identify if we have a pair or not. So, now that we have the attribute variable `strong_hole`, we can include a `RaiseAction` in our `get_actions` method. When raising we will want to know the min and max raise for the board we have a pair at. Once we have checked this we will just go all in if possible, and if we cannot we will go back to our check call strategy. [12]

We run the engine with this bot against the check–call bot and analyze the results. This time, we win against the check–call bot by a very large margin; this is an example of how an even basic strategy can help significantly.[13]

[12] The following code is on GitHub at mitpokerbots/reference-lecture-1-2021.

[13] Note that this strategy is deterministic, which is actually undesirable—in the next class, we will cover why. If you are playing purely based on how good your hand is, your opponent will be able to tell and then dominate you.

2 Lecture 2: Poker Strategy

This lecture is taught by Stephen Otremba[1] and Andy Zhu.[14] All code from lecture can be found at the public GitHub repository mitpokerbots/reference-lecture-2-2021. The slides from this lecture are available for download on GitHub and Canvas.

At this lecture (and all others), we raffle off a pair of Sony Headphones. This lecture, we also raffled off an Amazon Echo Dot. Attend lectures in person if you want a chance to win!

We'd also like to thank our 10 Pokerbots 2021 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at pkr.bot/drop to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Next, a reminder to check out Piazza Post @5 to find potential teammates.

Finally, our TAs recorded an introduction to Python and posted it in Piazza Post @36. This post also contains resources on introductory Python, and our course staff is happy to help in Office Hours![15]

2.1 Hand Types

A good way to understand hand types is by looking through examples. Slide 16 shows a board on the turn, with a 2c, Kd, Th, and 2s showing. Regarding hand notation, cards are described by value (2 through K) and suit (clubs, diamond, hearts, and spades). Note that it is impossible to have a flush on this board because regardless of the unknown fifth board card, at most two cards on the board could share a suit, which allows for at most four cards sharing a suit if the two private cards also match (five cards sharing a suit are required for a flush). Also, a pair of 2s is showing on the board—a board that has a pair on it should be played differently, as the baseline is stronger (everyone has a pair or better). The hand Jc and Qc in this scenario is called a "drawing hand"; once the fifth and final board card is drawn, the board will either be really good or really bad for us. This provides us with some certainty on the river, which is very good to have in an uncertain game like poker. We could get a straight (with either a 9 or an A), so we have an "open ended straight draw" right now (there are two ways to get a straight with the river). If we only had one way to get a straight, it would be a "closed straight draw", or "gutshot draw." Since 9 and A give some certainty of winning, these are called "outs."

Now, lets compute hand strength. Hand strength is computed on a scale from 0 to 1, and it measures how many hands out there will beat our hand. The graph on slide 17 gives us our hand strength in this scenario as a histogram, taken over all the 46 possible rivers. The likelihood of us having a particular hand strength corresponds to the area of the histogram bucket. If we get a 9 or an A, then our hand strength will be about 1, which is why there is a bump on the graph around 1.0. Similarly, the two small bumps above 0.5 and 0.6 in the graph correspond to us drawing a J or a Q respectively to give us a two pair. Note that this graph is very bimodal—this supports our intuition that a drawing hand provides avenues to either a very strong or very weak hand. The bimodal distribution is something we like to see, as it will settle to a point mass that has clear hand value by the end of the round.

Now, looking at the hand strength graph, you might notice that we have a large probability of having a losing hand with strength $\sim$0.4—this corresponds to us not hitting anything that will improve our hand on the river.

Slide 18 has another example scenario, this time with pocket 3s instead of the J and Q. This hand already gives you a two pair, which is better than nothing, as the board already has a pair; however, this hand loses to a lot of possible hands, such as any higher pocket pair, a K, or a T, since this would make a higher pair with the board. This hand does beat nothing though, which is not too bad. Looking at slide 19, we see the hand strength graph for this pair—note that this graph looks very different than the one on slide 10. In Texas Hold'em, a player has roughly a 50% chance of getting a pair or better on a random draw, explaining why there's a bump on the distribution around 0.5. You might also notice that there is a bump around 1.0—this will happen if the last card is a 3, which would give us a "full house," one of the best possible hands on this board. A problem with this board though, is that even once all 5 cards are revealed on the board, we still don't know if we're winning or not—we don't know what the opponent has. This is different

[14] Email: andyzhu@mit.edu.

[15] The Office Hours schedule can be found in Piazza Post @11.

from the drawing hand, where we know exactly whether we'll likely win or not based on the fifth card. Therefore, even though the low pair has more mass to the right than the drawing pair, its uncertainty means that it'll make us lose money more often because of how our opponent can fold if they have a bad hand or keep playing if they have a good hand (we are "adversely selected" against). When you bet high with a low pair, there aren't many opportunities for improvement, and you have a good probability of losing—you should be skeptical of playing longer when you have a hand with a distribution like that of the low pair.

Next, we'll talk about the "made hand" (slide 20). We'll talk a lot about hands that are good, because when you have a bad hand it's very easy to fold out. That's why the best pokerbots will be the ones that can differentiate good hands from better hands from the best hands, because they will know when to play and how. They're able to bet cleverly to extract the maximum value from their opponent. The made hand is also a two pair, and the K is called the "top pair." Even if the opponent has a T, we'll still win the two pair when it comes to hands, and we also have a good fifth card (the A), which is called the "kicker" or "tiebreaker" since it beats most cards.

Notice that even though we still have a two pair, the made hand has a very different hand strength from the low pair (slide 20). No matter what shows up as the fifth card, there are a lot of hands that our hand will strictly dominate, or win, in any scenario. In a game like poker with a lot of uncertainty, this is exactly what you want—and so the made hand is considered a good hand that will make us a lot of money. The reason why there is some mass around 1.0 is because we could get a full house if another K shows up. Note that there are still some hands that could beat ours (pocket As, for example). If you run into a scenario like this you could lose a lot of money thinking you have a great hand but run into an even better hand. However, in the long run betting there is net positive gain.

The last hand we'll be talking about is "the nuts" (slide 22); when we talk about the nuts, we mean that there exists no better hand than ours on the current board. When you have the nuts, you want to extract as much money from your opponent as possible; you want to bet, raise, or call every possible turn. The distribution for the nuts (slide 23) is basically 1: it is a point mass. In this case, by the turn the nuts are 2h 2d. We are simplifying this calculation a bit, as there are exactly two hands that would beat this: the opponent has pocket K's and the fifth card is a K, or the opponent has pocket T's and the fifth card is a T. This is an extremely unlikely scenario, one where both players have a four-of-a-kind but the opponent has a better one. If this scenario occurs and strong hands go against each other, both players will be aggressively betting and the pot will be massive.

Even though there exists a scenario in which our four-of-a-kind loses, that does not change the fact that *on the current board*, this four-of-a-kind is the best possible hand. There is a small possibility that the fifth card will change the board to introduce a better hand, but there are still no two cards we would rather be holding right now than a pair of 2s to match the pair of 2s showing on the board.

Again, remember that you want your pokerbot to focus on the hands that are good, as you'd often be better bluffing with a drawing hand than bluffing with nothing, even if you're confident in your bluffing abilities.

2.1.1 Board Types

In Texas Hold'em, there are also types of *boards* you must consider. A simple example is the one we've been using this whole time: a 2, K, T, 2 board. In the first scenario, you have pocket Aces (slide 25). This hand is much better than almost every hand, unless your opponent has a three-of-a-kind or better. You would feel great about your hand strength here. Instead, let's think about the scenario in slide 26. You'd almost certainly feel poorly about your hand strength here, even though you have the highest possible pair. The board has four clubs and several straight, flush, and straight flush possibilities. On a board like this, you can end up losing a lot of money to someone who gets an unlikely hand for an ordinary board, but a more likely hand in this "drawing board."

In Blotto Hold'em just like Texas Hold'em we almost always win the game in this situation. If we lose a game it is just a poor case of bad luck.

2.2 Pot Odds

Pot odds are very related to the idea of maximizing expected value. We're going to begin with a claim, which is that for any state of the game, there exists some probability of winning. Even if our opponent adopts a strategy that incorporates randomized behavior, this probability p still exists. Given that we have some probability of winning, we can calculate an expected value using the equation on slide 29. If we continue to play, the expected amount we'll win is $p \cdot$ `pot.grand_total`, and the expected amount we'll lose is $(1-p) \cdot$ `cost_to_continue`. Note that these amounts aren't symmetric. That's because in poker, you should consider every cost a sunk cost; that is, never worry about money committed to the pot in the past, as it is already gone. The only cost you're considering is the further cost of continuing, which you're using as some stake to win all the money in the pot. When it comes to expected value, we don't want to make a negative expected value decision - in fact, we wish to maximize expected value. Note that if the expected value is 0, we're indifferent when it comes to folding or continuing.

The next step is to perform some algebra to separate out the probability of winning, and this gives us a cutoff for whether or not we stay in the game. We call the right hand side of the new inequality for p the "pot odds:"

$$\frac{\texttt{cost\_to\_continue}}{\texttt{pot.grand\_total} + \texttt{cost\_to\_continue}}.$$

If we know our opponent's strategy well enough to calculate p, we'll never have to worry as we can always calculate pot odds to make a positive expected value decision against every bet.

Now consider an example for calculating pot odds (slide 31). We mentioned this example briefly, but we did not consider pot odds. On average we'd expect our opponent to win this board, as they have a made hand and we're banking on a straight to win. Remembering our distribution, only ∼2 out of 13 cards will complete our straight, but our opponent already has a good hand. Suppose our opponent puts 10 more chips into the pot, and we now have to decide whether or not to continue. Let's calculate the pot odds using the previous formula (slide 31). If we think our probability of winning is greater than 0.1, we should continue, and otherwise we should fold. This explains why drawing hands are so much better than you might otherwise expect; estimating our probability of winning is easy, so it's easy to make good decisions with them. Our opponent gave us pot odds that look good enough for us to stay in with our drawing hand and make money on average, so they "underbet." This is highly undesirable in poker, because it will let your opponent stay in the game when they should've folded a long time ago.

Let's look at this from our opponent's perspective. If our opponent made a higher bet that caused us to fold, they would've won 80 chips 100% of the time. However, with us staying in the game, they will win 90 chips ∼85% of the time, for a win of ∼76.5 chips on average. They are worse off by underbetting!

This is a little unintuitive to people unfamiliar with poker; you might think that you should just bet proportionally to hand strength, but you should generally not underbet as this would give your opponent opportunities that they would've otherwise not had. In addition, it reveals too much about your hand. If your opponent had instead bet more confidently, you would've folded as your pot odds would've looked a lot worse. For example, if your opponent had gone from 40 to 80, your pot odds would be 0.25, which is not good enough to merit a call.

There's also something called "reverse pot odds," which unsurprisingly, are pot odds for your opponent that give your opponent the opportunity to call. When we talk about reverse pot odds, it means that we're considering whether or not our opponent will stay in the game. The way that we can give our opponent opportunities is by "overbetting" relative to the size of our pot, which will give our opponent the possibility to exploit the pot odds and take our money. Overall, when considering pot odds you gain much more control over the size of the pot and maximize expected value.

Now, we'll consider an example bot that you may have seen on the scrimmage server: the "all-in bot." Our opponent goes all-in before the flop which is a deterministic strategy. Note this is easy to beat, we can check-fold, letting our opponent collect the blinds until we are dealt a high pair to crush them and win big.

2.3 Ranges

When we're faced with a bet, everyone knows the pot odds. If we can estimate p better than our opponent, then on average we'll make money. Ranges are the types of hands that you play—when playing poker, you

want to be restrictive about the hands that you play. Our opponents *range* is the distribution of hands we expect them to hold. We can estimate this during the course of the game—if our opponent hasn't folded late in the game and has bet a lot of money, we'll expect them to have a better distribution of cards. This means that ranges are key to calculating our probability of winning, which affects pot odds and our decisions. When it comes to poker, the best play style is typically "tight-aggressive," which means folding early and often with bad cards and betting aggressively with good cards. If your strategy is more tight-aggressive than your opponent's strategy, then you will often win more money on average, because you will get into high-stakes scenarios with better cards. For regular poker, the "tight-aggressive" strategy is folding ∼70% of hands preflop and betting frequently with the rest.

2.4 Variant Strategic Considerations

We are dealt 6 cards which means you have more information at the beginning and you can form more cards, which mean more pairs and drawing hands. People will generally 'hit' more since they are making hand types and possibly make 'throwaway' hands for boards with the smallest blind. Overall, your 'out' calculations change as you have much more information than you typically have in normal Texas Hold'em. When allocating your hands to your boards you should adjust the expected ranges. Note: It may not always be the best to play JJ into the best board as you can run into a lot of J+ (Jack and another face card)

When you have placed your bets you will want to control the pot and be able to manage each board the best you can. By giving better pot odds to different pots you can control the pot size. Now, if one of your boards is super good, try putting more money in there!

All-in Bots can be a quite powerful strategy with this variant. If you go all in on one board that means you go all-in on all your boards because there is no money to commit to the rest of the boards you are playing. Lets say you have allocated your best cards to 3BB and our strategy is to go all-in on that board. Then on the other 2 boards you win half the time, lose half the time (net change 0), making 7.5 chips each all in. Let's also say that if the opponent calls your all in, they probably have AA, KK, or another hand that is strong. AA, KK lose 20% of the time(ish). So let's say you have 20% chance of winning. This means that in showdowns, you expect to make (400+6)*20/100 - 200 = -118.8 chips. So, under these assumptions, this is a winning strategy if you can get folds 16 out of 17 times. This is assuming two things: 1. no extra money put in (this is likely false and probably brings it up to like 10 chips each all in, raising 30% of the time to 12 = + 3.6) 2. AA KK is top of their range, your probability of winning is probably better than 20% especially if you have an A. *Note*: All-in Bot strategy was not discussed in lecture.

2.5 Coding `reference-lecture-2-2021` Bot

We'll build off our `reference-lecture-1` bot, where we created an all-in bot when we recognized a pair. We start by creating a list of length three, `[0,0,0]`, and initializing a variable to be called `self.hole_strengths`. From this we then want to make a new method called `calculate_strength`, which will run a Monte Carlo process to simulate games and determine the win rates of our cards; in turn helping us determine how we raise, call, and fold, and which cards we want to assign to specific boards.

At the very beginning of our method, we want to first evaluate each of our hands; we do this by using `eval7`, a poker hand evaluation package in Python. We call `eval7.Deck()` to construct a deck that we will use for simulating multiple rounds of poker, and start by removing the cards that we have from this deck as they cannot appear again. Going through the numerous iterations of simulated potential hands, we then declare variables like `_COMM` and `_OPP` to represent how many community cards we can draw and how many cards each player has on that board pre-flop, respectively. To evaluate the strength of our hand, we will assign it a score (initialized to 0) representing its win frequency. To denote a win in our simulation we add 2 to our score, and to denote a tie we add 1; losing does not impact our score. After all iterations have run, we may now divide our score by twice the number of iterations to yield the calculated strength (or win probability) of our hand, stored in `hand_strength`—which we will use later on.

Note that working with Monte Carlo simulations can be very time-intensive, so we will want to limit the number of simulations we run. Accordingly, in this case we decide to run 100 simulations—as you can see in the `handle_new_round` method.

Now that we have set up our Monte Carlo simulation and the number of rounds we want to run, we will want to use this in our `get_actions` method. Here we just want to see how much we will want to raise by, and for this example we want to play a little more conservatively pre-flop. This is shown by us multiplying `(pot_total + board_cont_cost)` by a factor of 0.4 as opposed to raising the stakes deeper into the game. Once we do this, we will want to check how much it costs to make such a raise, and then commit the action depending on if we have enough chips in our stack.

Finally, we get to the part where we implement our hand win probability from the Monte Carlo simulation. But first, if our opponent raised on a board, we expect them to have a good hand; thus we will want to tone down our strength by an intimidation factor, as defined in the equation

$$\max([0, \texttt{strength} - \texttt{\_INTIMIDATION}]).$$

If our strength is greater than or equal to our pot odds then we may want to raise, and if not then we will want to fold.

Comparing the results of this bot to `reference-lecture-1` yields a large chip lead over our opponent, demonstrating the power of poker theory in improving your bot.

Finally, a reminder to always commit code back to git![9] There's nothing worse than ending a marathon coding session with a hard drive failure and losing all your progress.

3 Lecture 3: Game Theory

This lecture is taught by Andy Zhu.[14] and Haijia Wang[16] All code from lecture can be found at the public GitHub repository https://github.com/mitpokerbots/reference-lecture-3-2021. The slides from this lecture are available for download on GitHub and Canvas.

At this lecture (and all others), we raffle off a pair of Sony Headphones. Attend lectures in person if you want a chance to win similar prizes like these!

We'd also like to thank our 10 Pokerbots 2021 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at pkr.bot/drop to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

Finally, a reminder that Mini-Tournament 1 is tonight! Be sure to **submit your bot by 11:59pm EST on 01/8/2020** to receive credit for this course, the winner of this week will win $1000.[17] If you're pressed for time, feel free to incorporate some of the code we wrote in lectures 1 or 2, available at our public GitHub repository.

3.1 Pre–Lecture Game Rules

Before the lecture formally begins, we have two games for you to play. The first, which will be today's giveaway, is everyone submits a number between 1 and 1000 inclusive, and the person who guesses closest to $\frac{2}{3}$ of the average wins.[18] The second is opt-in or opt-out; you'll wager 20 ELO points, and the winner is the person with the highest last 4 digits of their MIT ID—they'll be receiving everyone else's ELO points.

3.2 What is a game?

There are many kinds of games, but we will only consider two–player zero–sum games. Two–player, very naturally, means there are only 2 players. The technical definition of a zero–sum game is one where any value won by you is lost by your opponent, and vice versa; there is no value created by winning the game. This is very naturally related to the concept of chips in poker.

A *game* between players 1 and 2 consists of a pair of strategy sets S_1 and S_2, and a utility function u. Examples of possible strategy sets are your bot going all in, or only check–calling. The utility function u outputs R, the utility for player 1, so the negative of this is the utility for player 2. Note that strategies are submitted simultaneously. Naturally, player 1 wishes to maximize u, and player 2 seeks to minimize u. This notation works very well for two–player zero–sum games, as each player is simply trying to maximize their own utility function. Examples of these games are: rock, paper, scissors (referred to "RPS" going forward); "my number is bigger than yours" (a game where two players say a positive integer and the one with the largest integer wins, referred to as "MNIBTY" going forward), and even chess or heads–up poker.

3.3 Pure and mixed strategies

RPS has 3 pure strategies: always rock, always scissors, or always paper. A mixed strategy is any that uses randomness to interpolate between pure strategies (e.g. flip a coin, and heads implies scissors and tails implies rock). If you're using a mixed strategy, for example, $p(rock) = .4$, $p(paper) = .3$, $p(rock) = 3$, you must choose one of the three pure strategies for a specific round. The reason we analyze mixed strategies is that a lot of fundamental game theory is only truly applicable when we let people use mixed strategies, choosing between various pure strategies. A pure strategy in RPS is not good, as it is easily exploitable if your opponent knows the strategy (e.g. always rock is beaten by always paper).

One of the many ways to represent games is through a matrix. The rows correspond to the pure strategies player 1 is allowed to take, and the columns to the pure strategies player 2 is allowed to take. The payoffs we usually write represent the payoffs to player 1. RPS is also easy to create a matrix out of, because each player has a finite number of possible strategies at every stage of the game—thus, it is a matrix form game.

[16] haijiaw@mit.edu

[17] Bots must be submitted to our scrimmage server at pkr.bot/scrimmage.

[18] This is called the "Beauty Contest" because there was initially a newspaper advertisement where you had to rank people's attractiveness, and the winner was the person with $\frac{2}{3}$ of the average rank.

Doing this allows us to compute the utility of a strategy S_1, as we can average payoffs of the pure strategies using the probability of picking such a pure strategy. We use $+1$ to represent the payoff of winning rock paper scissors, and -1 to represent losing; note that the magnitude does not truly matter as long as we are consistent.

	Rock	Paper	Scissors
Rock	0	-1	$+1$
Paper	$+1$	0	-1
Scissors	-1	$+1$	0

Another thing about RPS that makes it special is that it is a symmetric game, as switching the players or payouts doesn't affect the strategies.[19] This can be seen by the matrix as its transpose (switching the players) equates its negative (switching the payouts).[20]

3.3.1 The Keynesian Beauty Contest

Now, we're going to talk about the first game that we played. This is also a matrix form game, as each player has 1001 possible strategies; however, it has a multidimensional matrix, as there are multiple players each with their own strategies.

We'll analyze "rational" play for this game. The first thing that you notice about the game is we're trying to guess $\frac{2}{3}$ of the average, and the maximum possible number is 1000; therefore, whatever we play, we shouldn't pick above $\frac{2}{3}$ of the max, or 667. So, if no one is going to play over 667, the average won't be higher than $\frac{2}{3}$ of this, or 445. By the same logic, we shouldn't guess more than $\frac{2}{3}$ of this number. Continuing in this fashion, no rational player should play anything other than 0 or 1, depending on how you round. Notice how "rational" here is in quotes—this is because you're trying to win the Sony headphones, and not necessarily play rationally, as this does not always translate well to practice. This demonstrates why there's a distinction between playing game–theoretic optimally, and playing to win Pokerbots—something interesting to keep in mind while developing your bot's strategy.

3.3.2 Dominance

Next we'll talk about dominance: we say that strategy A "dominates" strategy B if playing A is always a better idea. In the Beauty Contest, submitting 667 dominates the strategy of submitting 1000, as we multiply the average by $\frac{2}{3}$. We can take this idea further by talking about "second–order dominance;" once we've crossed out the first dominant strategy, we can do the same consideration again. This is where we multiplied 667 by $\frac{2}{3}$ again to get 445. You can recursively apply this definition to even get to "infinite-order" dominance, which is when you only play strategies that are not dominated to any order.

This takes us to a point where no strategy can dominate another any more. It's helpful to define the point of an "equilibrium:" an *equilibrium* is a set of strategies, one for each player, such that nobody has an incentive to switch.[21]

Lets' look at a quick example (slide 18). If player 1 plays A and player 2 plays C then there is a positive output of +3 for player 1 and -3 for player 2. In this case this is the worst possible output for player 2. So, if player 2 instead plays D the they will only lose 2 instead of 3. So, it is in their best interest to always go with D if player 1 plays A. Now lets look at the second row, player 1 playing B. Without going into the examples we can see player could either get +1 or -1. Compared to going with option A where they can either get +3 or +2 it will always be their best interest to play A instead of playing B. As we discussed earlier if player 1 plays A then player 2 will always play D. Seeing this we then know there is a Nash Equilibria at (A,D).

3.4 Nash equilibria

You've probably heard the words "Nash Equilibrium" thrown around before. It may be very easy to get confused and think that a Nash Equilibrium refers to a perfect strategy; this is incorrect. A Nash equilibrium

[19] Note that being a symmetric game does not imply that its matrix is symmetric.

[20] This is referred to as the matrix being "skew–symmetric."

[21] Note that by "strategies," here, we are talking about mixed strategies. The "incentive to switch" means they cannot increase their expected utility by modifying their mixed strategy in any way.

simply means a set of strategies such that no player has a point in switching, but this does not always make it the best strategy. You will see an example of this when we go over the beauty contest statistics.

A famous quote, that many of you may have seen in *A Beautiful Mind*, is the guaranteed existence of a Nash equilibrium in every finite game using mixed strategies. This was proven by Nash himself, in 1951, using topology.[22] Note that this theorem only applies to finite games, which MNIBTY is not—for every strategy you can shift yours over to beat your opponent. We are also only refering to games as we defined them, where they are matrix form, you have a set of strategies for each player, and a deterministic utility function which given these strategies assigns a utility for each player.

A Nash equilibrium happens when everyone plays 0 in the beauty contest game (definition on slide 13). Even if you know that everyone else is playing 0, 0 is still $\frac{2}{3}$ the average of 0 so you have no incentive to switch, nor does anyone else. In the game RPS, there's no pure strategy equilibrium, as for every pair of player strategies players will be incentivized to switch to the pure strategy that dominates their opponent.

There is an "equilibrium" for RPS, which is playing each pure strategy $\frac{1}{3}$ of the time (slide 22). Let r, p, s be our probabilities of playing rock, paper, and scissors, respectively. Slide 18 shows the calculation for why both us and our opponent playing this same strategy is an equilibrium, as neither of us has an incentive to switch. Our opponent wants to pick a strategy r, p, s which minimizes our utility, and we want to maximize our utility given that our opponent is attempting to do this. The solution to our min–max relation guarantees us at least 0 value, which you may think isn't very good, but this means that on average we are never losing. If a pokerbot was able to get this sort of performance in a tournament, it would almost guarantee them winning as they would never be losing money on average.

Now we'll analyze a more complicated game, themed around military battle plans (slide 24).[23] I can choose to either charge or sneak attack against my opponent, and they can respond with either a full defense or defend in shifts. The matrix below displays my utility for each case:

	Full Defense	Defend in shifts
Charge	0	+3
Sneak attack	+1	−1

Looking at this matrix, there are mostly positive numbers; thus, before mathematically analyzing, we'll intuitively expect player 1 to have positive utility on average. Also, this matrix is not skew–symmetric, as switching the players and strategies changes the payoffs more than just multiplying by negative 1 (as one can easily tell by the +3 payoff).

Let c, s be the probabilities of me launching a charge and sneak attack, respectively. My expected values for utility as a function of my opponent's strategy then becomes:

$$\mathbb{E}[\text{opp full}] = c \cdot 0 + s \cdot 1 = s,$$

$$\mathbb{E}[\text{opp shifts}] = c \cdot 3 + s \cdot (-1) = 3c - s.$$

Let's suppose we want to find the values of c and s so that we are always guaranteed a certain expected value in this game no matter what our opponent picks—that is, we want to find c, s such that $\mathbb{E}[\text{opp full}] = \mathbb{E}[\text{opp shifts}]$. Calculating, we find that $c = 0.4, s = 0.6$ satisfies this constraint; substituting into either expected value equation tells us that no matter what our opponent picks, we have an expected utility of 0.6. Hence, the "value" of this game is 0.6 for us, or we are claiming value by playing this game.[24,25]

3.5 Applications to poker

I've shown you that Nash equilibriums exist, which may sound very weak for practical applications, but this information is very useful for finding it in different games; now let's look at it for poker. First, we have to

[22] The specific theorem is called the fixed point theorem, which some of you mathematicians in the room may have heard of.

[23] This game is themed around military battle plans, as game theory was actually developed significantly by military strategists at the RAND Corporation, a large think tank, during the 1950s!

[24] Note that if you use this value to guarantee a minimum expected value of utility in other games, you can get nonsensical numbers if dominated strategies are present.

[25] You can always use this computational method, even when more strategies are available, using linear programming for those of you who are familiar with it.

prove that poker is a finite game, but then lets talk more about what equilibrium actually means in the context of poker. RPS is very easy to represent as a matrix, but other times its more natural to represent games as a tree. This representation may be more natural to you if you've taken computer science classes using graph theory.

We'll look at the concept of "extensive form games;" we can represent a game that's played sequentially, such as poker, by game trees (slide 28). Consider the game of tic-tac-toe, which has its tree displayed on slide 29. You'll notice that tic-tac-toe is very symmetric, so we've made the tree smaller already—however, its tree is still very large. An extensive form game can be represented by a finite (possibly large) game tree where the nodes represent game states, levels of nodes are alternating players, and eventually you reach a leaf which is a number representing player 1's utility, rather than another game stage. Poker is an extensive form game, where you're using the information your pokerbot gets from the `GameState` to build your game tree. It's clear that chess is another extensive form game, but with chess the game tree is incomprehensibly large compared to tic-tac-toe's. These game trees can get really big because they grow exponentially, especially permutation poker with its combinatorial explosion; however, they can be dealt with through a technique called "backwards induction."

Looking at tic-tac-toe's game tree, we can start at some point late in the game and consider a specific section of the tree (slide 32). Note that the leaf nodes have their player 1 payoffs marked in blue. Next, note the black number next to certain nodes—this represents from that game stage onward, only the black number is possible as a payoff. We can do this by going upwards from leaf nodes, calculating what a player playing optimally would rationally choose between the game stages available to them at the level below that state. This upwards traversal means that eventually we'll reach the start node of the game, giving the game itself a payoff for each player. This approach has been applied to games as big as Connect Four, which is strongly solved, but not much bigger than that.

Every extensive form game—such as tic-tac-toe—is also a "normal form game," meaning that it can be expressed in matrix form. The problem with writing this matrix, however, is that it grows exponentially in nature, because the number of strategies for each player is equivalent to every possible non-end-stage of the game. This matrix in fact is doubly exponential, as the number of both row and column strategies is exponential. This existence, however, means that tic-tac-toe has a Nash equilibrium in existence.

However, there's a problem–we're still not working with poker! Poker has an unimaginably large doubly exponential matrix, but since all bets are in integral amounts of chips, it has a finite matrix. Therefore we know the existence of a Nash equilibrium for poker. There is one additional caveat for poker, however: the reason we can't explain poker with this type of game tree is because poker has incomplete information. With poker, you can have a situation like the one displayed on slide 36. Nodes in the poker game tree can be reached not only by player actions, but also by "nature:" elements of randomness. When the engine tells you there are three new cards and that's what the flop is, this is randomness, as well as when your opponent flips their cards over. Randomness is the following tree representation of our poker game is represented by player R. The dotted lines on the game tree explain that players don't know which node they're situated at because of what hand their opponent has, which is why the game tree becomes imperfect. R always leads to a node labeled 1, which represents player 1's turn. They then have two options (to fold or to bet), which leads to a player 2 scenario. One of the key aspects of poker is that our opponent does not know our hand—that is our secret knowledge—so this is represented in the game tree by our player 2 nodes being linked by a dashed line. Our opponent cannot tell whether we were dealt the AA or the AK, so the game states are effectively equivalent to our opponent. They can see our game tree movements, but not our particular game state. This is why poker is such a beautiful game to play. The question then becomes whether we can still apply backwards induction to these types of game trees; the answer is we cannot, due to these dashed lined which we refer to as "information sets." These dashed lines throw off the backwards induction process.

Poker is also a matrix game, as for every type of strategy that our opponent takes—betting, calling, or folding, for example—it's possible for us to create a responsive strategy based on the information that we have. This shows that Poker does have a Nash equilibrium after all, since it can be expressed in matrix form! This equilibrium strategy will have average payout 0, which will guarantee that we do not lose on average—that is, we cannot lose; however, this matrix is double exponential in size. Thus, computing the Nash equilibrium is near–infeasible due to computational complexity. Methods do exist to do this using linear programming, however, but even then it's still very difficult due to complexity reasons.[26] The answer

[26]The MIT course 6.853, "Topics in Algorithmic Game Theory," covers many ways to find and approximate Nash equilibria;

is also no for a different reason. If some of you were there for our pre–registration game Blotto, you may realize that game also has a Nash equilibrium; however, in Blotto, the optimal strategy is not to solve for the Nash equilibrium but rather to play against the other players. Similarly, in poker, your goal is to beat the other players, not solve for the Nash equilibrium. If you cannot solve for the Nash equilibrium, that is totally okay; your goal is to take your opponents chips. If you can play strategies that beat multiple opponents and take their chips, then you have a great shot at winning this tournament. If you target specific opponents, we call this an "exploitation strategy;" sometimes, these can do better than approximating or solving for the Nash equilibrium.

In poker, one strategy you can attempt is to approximate the Nash; this strategy will do pretty well on average. In practice, playing the Nash equilibrium in RPS is not always the best idea. For example, if you play an opponent who uses rock with probability 0.6, you want to play paper more than the Nash equilibrium suggests you do. The Nash will guarantee that on average you play the value of the game, and in each of the subgames in the game tree it will also have you play optimally in that situation. What this means is that you could definitely come up with a strategy that is not the Nash equilibrium that plays equally against the Nash, but finding it is computationally infeasible. The Nash equilibrium in poker, on the other hand, will always do pretty well, but might not be the best strategy against certain strategies which are really suboptimal that you could crush with a different strategy—it's up to you to determine when to approximate the Nash and when to do something different that makes your bot great rather than good.

Examples: Applied Game Theory

YouTube links:

1. https://www.youtube.com/watch?v=hRIXXCe0Hi0
2. https://www.youtube.com/watch?v=WqP3fj3nvOk

3.6 MIT ID Game Discussion

Just like the beauty contest game, we're going to analyze rational play for the MIT ID game. There's again a cutoff—this time, people would rationally only play if their last four digits are in the 90,00+ range. Again, however, people play to win, not necessarily rationally. People will always behave in ways that are hard to model mathematically, but it will be advantageous for you if you can figure out how to exploit this behavior in some way. This game is certainly one where it makes sense to deviate from the Nash equilibrium.

This game is a great example of "adverse selection," which happens anytime a "buyer" and "seller" have asymmetric information. You likely don't know anyone else's MIT ID, so you're only competing with people when they *want* to compete against you—selecting adversely to your chances of winning the game. There's a game called the "market for lemons"[27]: suppose everyone is trying to sell a car in a lot, where the cars vary wildly in quality. The prices range from \$1000 to \$10,000, and you're trying to determine the fair price for a car on this market. If the fair price is \$5,500, all cars worth above this value will leave the market as their sellers don't want to lose so much money. Now, the average value of cars on this market will drop again, and as a result the fair price will drop as well. This process continues until all cars left on the market are the worst of them all (the "lemons"). This happens because there's so much adverse selection—every seller has private information about the true worth of their car.[28]

There are many sources of adverse selection in the game of poker: e.g. choosing to bet on the first action, since your opponent is much more likely to call when they have good cards, etc.

we highly recommend it.

[27]The word "lemon" is slang for a car that is discovered to have problems only after it's bought. This refers to the fact that when you drive a new car off the lot it loses 20% of its value instantly. If you try to sell your new car the day after you purchased it, people will wonder why you're selling it since they have less information about it than you.

[28]A common economics joke is that if you find a \$20 bill on the street, you should pick it up; if you find a \$20 bill in the middle of Grand Central, you should leave it be since if all the rational people around it are walking past, there must be some adverse selection to picking it up.

3.7 Reference Bot

In the the prior lecture we were able to understand how strong how hands were using a Monte Carlo strategy. today we will modify of this a little bit by leveraging some of the Poker Theory we covered by creating strong hands and the Monte Carlo Strategy.

First we are going to do is define a method called `rank_to_numeric` which will help us rank the card. So we rank cards 10 - A with values of 10 - 14. Now there are 13 cards in the deck but since we are starting at 2 it goes from 2 - 14. Then we will make a sort-cards-by-rank method that will sort our cards in descending order.

Now that we have identified the the cards and sorted them lets implement this into our strategy. In our `allocate_cards` method we first look to identify strong pairs. The first for statement that we write looks to identify these strong pairs. We want to make sure we record that we are going to record a pair if it is strong, so we append the strong pair to the `holes` list and update our allocated cards.

The next hand we want to look for are straight draws. To review a straight draw is when we have 5 cards not all of the same suit. To if we had: 10A, JQ, QC, KD, and AA this would be a straight. To look for straight draws we look to see if the difference in value of the card is less than or equal to one and if they are unused. If so then we will want to use them! We will update our hole variable and update our allocated cards again.

Next we look for flush draws in a similar fashion as before. Where we go through our remaining cards and look to see if we have four cards of the same suit, so we would only need one more card of the same suit to have a draw, which is another powerful hand! We then go through our dictionary and see if we have either length 2 or 3 that have the same suit which we need in order to have a flush.

Finally, we go through the extra cars that we have and group them randomly and update the remaining cards after we group up the remaining cards. We also want to add two `assert` statements. Assert statements are a great way to catch error in your code and help you debug. So, in our two assert statements we want to check that we don't have any remaning cards left and check to see if we have allocated too many cards. After this we just return the decisions we have made by returning `holes_allocated`

Comparing the results of this bot to `reference-lecture-2` yields a large chip lead over our opponent, demonstrating the power of game theory in improving your bot. As always the lecture bot is posted at: https://github.com/mitpokerbots/reference-lecture-3-2021

4 Lecture 4: Performance

This lecture is taught by Stephen Otremba.[29] All code from lecture can be found at the public github.com repository mitpokerbots/reference-lecture-4-2021. The slides from this lecture are available for download on Canvas and at the public github.com repository mitpokerbots/class-resources-2021.

At this lecture (and all others), we raffle off a pair of Sony Headphones. Synchronously attend lectures if you want a chance to win them!

We'd also like to thank our 10 Pokerbots 2021 sponsors for making Pokerbots possible. You can find information about our sponsors in the syllabus and on our website, and drop your resume at pkr.bot/drop to network with them. Our sponsors will also be able to see your progress on the scrimmage server, giving you a chance to stand out over the course of our competition.

This week, we will be holding our guest lecture featuring Libratus creator, Noam Brown this Friday at 1 pm EST. Noam Brown is a Research Scientist at Facebook AI Research working on multi-agent artificial intelligence. His research combines machine learning with computational game theory. He is also well known for creating Libratus, an artificial intelligence computer program designed to play poker, which has decisively defeated top human poker professionals. We hope to see everyone there!

Moving onto class updates, we have a C++ update on the class Github repository. You can get the latest changes by navigating to your engine 2021 repository and running the git pull command.

Before we get into the winners of this weekend's past tournament, we have a brand new prize to announce. Beginning today, all students have the opportunity to begin to pursue the Strongest Alliance prize, which introduces an incentive to team up and make Pokerbots even more fun. The strongest alliance prize will go to the alliance that places highest in the next weekly tournament. In order to make an alliance, your team must merge with one or more other teams and begin making bots together for the rest of the class. In order to register this alliance, navigate to pkr.bot/alliance to establish a new alliance; this link will go live later today. If you are looking for new teammates, more info. regarding new team formation will be posted on the class site and Piazza.

4.1 Pre–Lecture Game Rules

Before the lecture formally begins, we have one game for you to play, which will be today's giveaway. You will submit a guess for the runtime of Tournament 1. The winner of the game is the person closest to the real answer in minutes.

4.2 Running Code

Today we will be covering the basics of code performance, including a brief introduction to algorithms, how to analyze them, and conclude with a live coding session to make our bot much more efficient.

The first question we ask ourselves is, "what happens when I run code?" And to put it shortly, it depends. It depends on what language (Python, Java, C++, etc.), what version of the language, what operating system, and what processor you are using. Evidently, there are a lot of factors to consider.

In general, running code begins with translation. Code needs to be translated into something the computer can actually read. This translation is done by a compiler or interpreter, or some combination of both. These translators convert code you just wrote to machine code. Machine code is the lowest level of instruction that your computer can understand. It is essentially a bunch of ones and zeroes that all have a certain meaning to your computer and your processor. Machine code is a set of instructions your computer will complete in order to execute all the logic you specified in your original program.

All these instructions are completed by your computer's core components. These are the lowest level systems of your computer that compute everything from math to memory management, and they are all invoked through these machine code instructions. Machine code is read and translated by your computer's processor, the CPU. The CPU translates the instructions it receives and determines which other computer components it needs to loop into the program in order to execute the different actions that you just asked it to fulfill. This action could be a math operation such as addition or division, which would be handled by the processor's arithmetic logic unit, known as the ALU. It could also be an instruction to remember something

[29] Email: sotremba@mit.edu.

simple, which might mean the processor will store some data in small memory buckets known as registers. It can also be an instruction to read some file that you have saved on your computer, which means processor might have to ask for data from the main hard drive or memory caches.

Though we now know what happens when code is run, we are still stuck with the question of what takes the code so long to run. Generally speaking it comes down to two things: one, the translation process of code (either by compiler or interpreter), and two, the instructions themselves and the processes they trigger in the computer. The process of translating into a different language can be very time consuming, and once that's over, these instructions can trigger time consuming processes.

First, let us talk about the translators. In computer programming languages that you are probably familiar with, this translation is either handled by a compiler or an interpreter or some combination of both. This fact is true for all the languages used in the Pokerbots competition. Although they complete the same function of translating your written code into machine code, these languages do it in different ways. At a high level, compilers begin by translating your code all at once before it is run by the computer. This means the compiler will scan the entire code file that was just written, convert it to a machine readable format, and once this process is completed, it will run all the code by sending the translated instructions to your Processor. This process is typical of languages such as C, C++, Java, and Go. In contrast, interpreters translate code a bit at a time, when it is bring run. An analogy would be laying down railroad tracks for a train that is running down them. This process is typical of dynamically typed codes such as Python and MATLAB. Again, these are very basic overviews of these two processes, and these two methods result in varying pros and cons

For compilers, going through all the code at once can be slow to complete, which is why some people may ask about compile times on Piazza. However, once that process is over, the actual execution of the code is incredibly fast. One other note about compilers is that compilers are usually limited to languages with static typing. On the other hand, interpreters read through and interpret the code as it is being run, which results in a slow process. However, we often have an easy time writing the code in this case because languages utilizing interpreters support dynamic typing.

With this information on hand, what does this information tell us about choosing a programming language? Beyond swapping languages, there's not much that can be done to help the code's translation process besides rewriting it or precompiling. Hence, we recommend choosing the language you feel most comfortable with. Other factors to consider are, if you need really fast code, try Java or C++. If you are new and are looking for an easier time programming, try Python. We encourage all of you to give learning a new language a shot.

Beyond considerations of language, the core instructions that are given to the computer can really impact the speed of the code. For example, addition, subtraction, and other math operations supported by the arithmetic logic unit are typically fast. In contrast, loading and moving variables in code can be slow. For example, loading and moving is fastest in registers, slower in main memory, and slowest in disk/storage. Evidently, we need to be clever with how we use these instructions, which leads us to the next section, efficient Algorithms.

4.3 Algorithmic Considerations

Algorithms are sequences of instructions we give to computers to solve a particular problem. Whenever we're faced with a problem in computer science, we typically make an algorithm to solve that problem! Essentially, an algorithm is a recipe we can follow to solve a type of problem every single time we need to. For example, a Pokerbot can be thought of as an algorithm for Blotto Hold'em! The Pokerbot will solve the problem posed by the Blotto Hold'em game. In this discussion of algorithms, it is important to note that not all algorithms are made the same. In the case of a Pokerbot, this might mean that some of them do the wrong thing in certain situations and might not solve the problem we really want them to.

One example of an algorithm is the solution to the following problem: given a list of numbers, sort the numbers. There are dozens of algorithms that solve this problem. As examples, these include Merge Sort, Quick Sort, Heap Sort, Insertion Sort, Bubble Sort, Counting Sort, Selection Sort, Radix Sort, Bogo Sort, Timsort, etc.. Despite all these algorithms, not all are created equal. Namely, some algorithms will solve the problem (sort the numbers) more efficiently than others. To better understand this idea, let us take a look at one of these algorithms, Bogo Sort. In Bogo Sort, in order to sort the list of numbers, we randomly

shuffle the list that we get, if it is sorted we return it, otherwise we shuffle it again. We continue to do the process until we find the sorted list. Evidently, this is not the best possible way to solve this problem. Hence, we ought to develop a way to compare these algorithms to determine which would be best. This idea is generally known as complexity, and it captures the idea of how complex and efficient our algorithms are. Generally, we are concerned with time and memory complexity. Time complexity refers to how long we expect our algorithm to take, while memory captures how much space on our computer we need to allocate to solving this problem. For now, we focus on time complexity.

When measuring time complexity, we may be tempted to measure in seconds, how long the algorithm took to run. However, in reality, this measurement may not be the most practical and useful. For example, an algorithm can run in 1 second on a new computer, but it may take 10 seconds to run on a old computer. This disparity is an example of what can happen when wall-clock time is used to measure run time of an algorithm to evaluate its efficiency. The exact set up of a machine has a large impact on how well a piece of code runs. In contrast, we seek a measurement that says the same thing despite which software or machinery is being used. Hence, we use asymptotic complexity, which answers the question, "how many operations does our algorithm perform as a function of our input size?"

With this idea of asymptotic complexity, let us look at a few examples. The next example is the searching algorithm, which is given a list of numbers of size n, determine if the number k is in the list. For example, say we have a list $[1, 6, 9, 3, 29, 12, 7, 8, 10]$, and we want to determine whether or not 7 is in the list. One potential algorithm to solve this problem is to check each element of the list, and if the element is equal to k¡ then we are done! If not, none of the numbers are equal to k, so the number must not be in the list. To evaluate this algorithm, we can determine how many operations or steps we must have taken to find an answer. If we consider our solution, we check each element and compare it to our value, k. In the worst case, we need to check every single element in the list. Hence, our algorithm might, in the worst case, take n operations. So in this case, we have n as a measure of how efficient our algorithm is.

Let us take a look at some more examples. Imagine another algorithm where we perform a few operations, such as 5, for every item in the list. Thus, we would have $5n$ operations in the worst case. In other cases, we could perform $7n$ operations for every element in the list, which would be $7n^2$ total operations! In all cases, we care about what happens only as n gets arbitrarily large, which is known as asymptotic complexity.

In asymptotic complexity, we use the limits of our complexity functions to classify our algorithms, Big-O notation captures this behavior in the worst case. We say the following,
A function $f(x) = O(g(x))$ if $|f(x)| \leq C * g(x)$ for all $x > x'$ for some x' and some constant C.
Essentially, $f(x) = O(g(x))$ if $g(x)$ grows at least as fast as $f(x)$ up to a constant.

Going back to our previous examples with this new understanding of asymptotic complexity, let's analyze the run-time for the same algorithms. In our sorting example, we have about n operations to perform. Thus, our algorithm is $O(n)$. If we had $5n$ operations like we did in the second example, our run-time would remain $O(n)$ because both n and $5n$ are the same up to a constant (5 in this case). Lastly, the example with $7n^2$ operations is $O(n^2)$ for the same reasoning.

Another set of operations is known as constant time operations. There are a few operations that computers perform incredibly fast. These operations don't depend on size for most uses, and are thus called constant time operations. We refer to the complexity of constant time operations as $O(1)$.

4.4 Python Data Structures

Using our acquired analysis of complexity, we can apply that same analysis to the Python language. We can translate this analysis into Python, as each Python operation is associated with some cost. Examples of constant time $O(1)$ operations in Python are the following:

- Math on numbers $(+, -, /, *)$, etc.
- Adding things to the end of a list (`append`)
- Assigning a new variable (`x = 5`)
- Getting a value at a list index (`x = list[i]`)

On the other hand, there are some operations that are not quite as fast. Examples of these include the following:

- `in` operator is slow when used with lists: $O(\text{size})$
- `for` and `while` loops can be big time hogs: $O(\text{loops})$
- copying things (especially lists): $O(\text{size})$

Given the high costs of these actions, there exist Python built-in data structures that help make our code much more efficient! Notably, Python has two main hash table objects, dictionaries and sets. Dictionaries map keys (which can be any immutable type) to values, and these structures allow us to look up a value using its key in $O(1)$ time. Additionally, sets keep track of a collection of items, and for sets, the `in` operator searches sets in constant time - much faster than lists!

Another Python tool that helps us make written code more efficient is Python libraries. Python libraries utilize faster coding techniques under the hood to achieve speed increases in practice! Essentially, users can utilize these libraries and their functions directly by importing them without having to worry about actual implementation, which is abstracted away. Below are some useful libraries that you may find helpful in coding your Pokerbot, which will also be utilized in building today's reference bot:

- `import pandas as pd`
  - Useful for storing, loading, and manipulating spreadsheets of data in Python
- `import numpy as np`
  - Great for operating on large arrays of numbers with amazing speed. Written in C behind the scenes
- `import itertools`
  - Used for efficient looping: combinations, permutations, counting, etc.

4.5 Reference Bot

With these new concepts on hand, let's take a look at how efficient our bot currently is and see how we can make it just a little bit better. Our main goals for this coding session will be utilizing more of the space that we are given for our bot (100 MB of compressed space to use). In particular, we will be trading some of the unused space for more time during the competition, which enables us to run more complicated functions later on. We will do this by running most of our inefficient code offline and saving the results in our bot.

Currently, Monte Carlo is taking the largest chunk of time in running our code. In order to estimate our hole strengths, Monte Carlo needs to be run 3 times per round of Blotto Hold'em. Since there are three boards of Blotto Hold'em and we play 500 rounds per game, there will be 1500 Monte Carlo simulations that need to be run every single game. Consequently, if this process is inefficient, it will ultimately hurt our overall time complexity, especially when we increase the number of iterations.

With our new algorithmic tools, let us quantify the efficiency of the Monte Carlo algorithm. Copied on the slide is the most time-intensive section of the algorithm. Looking at line 16, this is the line where we copy a whole deck of cards, which will take $O(\texttt{num cards})$ to complete. Next, on line 20, there are a few cards removed from the deck to make the deck more accurate, which will take $O(\texttt{num cards})$, as it is essentially using the `in` operator. The most time-consuming portion of this process begins on line 24, where we loop through whatever the size of `iters` is, which means this section is at least $O(\texttt{iters})$. However, inside this loop, on line 25, we see that we must shuffle the deck upon each iteration to preserve randomness, and we assume that this shuffling takes $O(\texttt{num cards})$, which is the number of cards. Hence in total, this loop will take $O(\texttt{iters}) \cdot O(\texttt{num cards})$ time, $O(\texttt{iters} \cdot \texttt{num cards})$, which is far more time consuming than our other processes.

To alleviate this issue, we notice that there are 52 choose 2, 1326 total possible hole cards in the game. For each possibility, we need to calculate its strength. Comparing this value to the number of times we have been running Monte Carlo (1500 calculations per game), we note that we are doing far more comparisons than we actually need to. We are calculating more hole cards than we can actually make with a 52 card deck! Evidently, we can benefit from storing some of these calculations over time for use later on. Additionally,

we can still do better. For example, if we get a hole card that is Ace of Hearts and King of Diamonds, our probability of winning with that hand is basically the same as if it were an Ace of Diamonds and King of Hearts because all suits of the deck come up with equal probability. Hence, we only need to consider suited (two matching suits) or off-suited (two different suites) versions of each hole, bringing our number of possible hole cards to 169.

For our reference bot, we will act upon these possible improvements by first making all pairs of ranks we can encounter in a game sorted by rank. For example, AA, AK, AQ, AJ, AT... and so on, KK, KQ, KJ, KT... and so on. For each of these holes we will consider them as either suited or off-suited. Recall that suited means the two cards are of two matching suits, while off-suited means that the two cards are of two different suits. Hence, pocket pairs are always off-suited. Following this categorization, we calculate strengths for an example of each of the holes, which as we have shown before is only 169 simulations. We perform all of these steps in a new Python file, `compute.py` offline and store the results in a `.csv` for quick access later. By running all of the most complicated operations offline, we are able to run more iterations of Monte Carlo to gain a massive improvement on our bot.