Pokerbots Course Notes

Massachusetts Institute of Technology
6.176: Pokerbots Competition[*]

IAP 2022

Contents

[*]Contact: pokerbots@mit.edu

1 Lecture 4: Performance

This lecture is taught by Matt McManus[1] and Haijia Wang.[2] All code from lecture can be found at the public github.com repository mitpokerbots/reference-lecture-4-2022. The slides from this lecture are also available for download at the public github.com repository mitpokerbots/class-resources-2022.

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 11 Pokerbots 2022 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 on Zoom. 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!

We also will hold a poker night study break this Wednesday on January 12th. More details will follow shortly on Piazza. We are also announcing our final event, including sponsor networking, which will be held on Gatherly on Friday, January 28th.

Announcing our Mini-Tournament 1 results, Grand Central was our winner with an ELO of 2124 while Ace of Spades took the biggest upset prize with a difference of 1449 in ELO.

1.1 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.

1.2 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 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.

[1]Email: mattmcm@mit.edu.
[2]Email: haijiaw@mit.edu.

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.

1.3 Algorithms

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 Swap Hold'em! The Pokerbot will solve the problem posed by the Swap 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.

1.4 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 $O(1)$.

1.5 Applications to Python

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.

1.6 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. Since we run 1000 rounds per game, there will be 1000 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 (1000 calculations per game), we note that we are doing far more comparisons than we actually need to. 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.