CS261 Winter 2018 - 2019
Lecture 1: Course Overview*

Instructor: Ashish Goel
Scribe: Kaidi Yan
Edited: Geoff Ramseyer

January 8, 2019

1 Overview

CS261 is the second course in algorithms. In this course, we deal with the following problems:

1. THE HOW - How do we solve problems efficiently? In particular, we are going to study the following categories of problems:
    (a) Linear programming (LP) / duality; flow problems; using LPs to derive general methods to solve decision problems
    (b) Sketching
    (c) Basic approximation algorithms
2. THE WHAT / THE WHY - What decisions should we make and why? Problems we are going to touch upon include:
    (a) Basic game theory, including zero-sum games
    (b) Multi-armed bandits, a general approach to sequential decision making

1.1 Course Logistics

1.1.1 Homework - 75%

There will be a total of 4 problem sets for this course, comprising 75% of your total grade. You are allowed to form a team of at most 3 people to work on each problem set. For each team you only need to make one submission.

*For Introduction to Maximum Flow, please refer to Tim Roughgarden's Winter 2016 CS261 lecture #1 notes on Introduction to Maximum flow

1.1.2 Final Exam - 25%

The final exam will make up 25% of your final grade, date and time TBA.

1.2 Three Motivating Examples

1.2.1 Counting Distinct Names - Sketching

Suppose that you are viewing a stream of names $a_1, a_2, ..., a_t, ...$, where each a_i is a string representing a name, e.g., $a_1 =$ *'Tom'* and $a_2 =$ *'Cathy'*. There can be duplicate names appearing throughout the sequence, and your task is to maintain an estimate of the number of distinct names appearing in the first T names. Your answer can be a (reasonable) approximation of the actual answer. How should you design an efficient algorithm to solve this problem?

If you have taken CS161, the first answer you might think of is using a hashtable using the names as keys. Suppose we have a total of N distinct names, and T is the total length of the stream. Then this algorithm will have a time complexity of $\Theta(T)$ and space complexity of $\Theta(N)$.

This method gives us the exact result to the original problem - however if N is large, it can take a lot of space. Is there an algorithm which has time complexity $\Theta(T)$ and near-constant space complexity, such that we can at least get a reasonable approximation result? In fact there is. We will see a formal exposition later in the class, but for now, we will provide an informal (and technically incorrect) explanation. We first define the following function:

```
function F($a$)
    $srandom(a)$
    return $random()$
end function
```

This function F takes in a name a, sets the random seed to be a and returns a random number between 0 and 1. Thus if $a_1 = a_2$, we are going to have $F(a_1) = F(a_2)$ since the random seed is set to be the same; on the other hand, if $a_1 \neq a_2$, then $F(a_1)$ and $F(a_2)$ will produce two independent random numbers between 0 and 1[1].

Now consider the following algorithm: compute $F(a_1), F(a_2), ..., F(a_T)$ as they appear in the stream, and as we compute these values, keep track of the maximum value out of all the $F(a_i)$'s. This algorithm takes $\Theta(a)$ space, assuming that we can store very high precision numbers, and $\Theta(T)$ time. If there are N distinct names, $F(a_1), F(a_2), ..., F(a_T)$ should produce N independent random numbers between 0 and 1 — it is easy to show that the expected value of the maximum value out of N independent random numbers between 0 and 1 is $N/(N+1)$. Hence we can use the maximum value returned by our new algorithm to estimate N.

[1]Technically, the random number generators are pseudo-random, and technically, just setting the seed does not guarantee a random number; we will address this more formally much later in the class.

Later in this course we are going to study several different types of sketches, of which the above is an example.

1.2.2 Two Slot Machines - Bandits

Suppose a gambler sees two slot machines and he wants to play one of them. He observed the first slot machines for 3 days and found that the first slot machine has 1 day of success and 2 days of failure. His friend, who has played the second machine for 2000 days, tells him that the second machine succeeds for 1000 days and fails for 1000 days. Which machine should the gambler choose to play in order to have a higher reward?

Based on the observations, we can claim that the success rate for the first slot machine is $1/3$, while the success rate for the second machine is $1000/2000 = 1/2$. Clearly the second machine has a higher success rate, and thus it seems intuitive that the gambler should choose to play the second machine in order to have a higher expected reward.

However, what if the gambler can play for many days? Given that we have only observed the first machine for 3 days, it might be worth exploring the first slot machine for a few more rounds to get a better estimate of its true success rate. Suppose that the gambler chooses to play the first machine. On the first day, based on past observations, the gambler would expect a $1/3$ chance that the machine succeeds. If the machine succeeds on the first day, then the observations for the first machine become 2 successes and 2 failures. Hence the observed success rate is now $2/4 = 1/2$. Suppose the machine succeeds again on the second day. Then the observed success rate would become $3/5$. It is possible that the machine just happened to fail on the first few days, even if its true success rate is higher than $1/2$. By exploring the first machine, about which the gambler knows little, the gambler can potentially gain increased rewards later on.

In this example, we faced the decision of finding the right balance between *exploration* versus *exploitation*. Later in the class, we will formulate the multi-armed bandit problem, which is a canonical model for thinking about these kinds of scenarios. And we will see how to design optimal polities as well as the implications in the multi-armed bandit problem.

1.2.3 NBA Games - Max Flow

Suppose that we are in the mid of an NBA regular season - each NBA game is played between two teams and there is exactly one winner and one loser for each game (there is no tie). Each team has played several games and there are more games to be played between teams. We want to know if there is any chance the Los Angeles Lakers (one of the NBA teams) is able to win the regular season.

One simple way to solve this problem is to enumerate alll possible results of the remaining games and see if there exists such a combination of results in which the LA Lakers will win the regular season. Howeve, if the number of remaining games (N) is large, then enumerating all 2^N remaining possible games can take a very long time. We want a more efficient algorithm to solve this problem - in particular we want an algorithm that runs in polynomial time.

In fact, we can convert this problem into a problem of finding the maximum flow from one vertex to another in a particular graph, which can be solved efficiently in polynomial time. We will see this reduction, and many other beautiful reductions from combinatorial problems to max-flow problems, later on in the course.

The max flow problem is one of the oldest combinatorial problems that computer scientists have studied. We will study several very beautiful (and also practically viable) algorithms for solving this problem. For those of you who are interested, it is still a major research question to find the fastest algorithm for solving max flow.