St. Francis Xavier University
Department of Computer Science

CSCI 541: Theory of Computing
Lecture 5: Probabilistic Computation
Fall 2023

1 Probabilistic Turing Machines

When we defined and compared deterministic and nondeterministic Turing machines, we noted that using nondeterminism in a computation could provide us with desirable behaviour; instead of deterministically following the same steps and ending up with the same outcomes, we could "make guesses" at certain branching points in the computation and see what outcomes arise as a result. By running all of these computation branches in parallel, we can evaluate every possibility at once to see if any branch produces an accepting computation.

However, despite how appealing this behaviour sounds, it doesn't actually give us any additional power to compute more things. Any nondeterministic computation can be simulated by a deterministic Turing machine with some penalty incurred in the amount of resources used. Indeed, since real-world computers behave in an entirely deterministic manner, we can't implement *true* nondeterminism in any algorithm we write—it is, in effect, an academic notion that we use to reason about aspects of computability and complexity.

Since nondeterministic behaviour is desirable yet we can't implement it directly, we must settle for some approximation of nondeterminism, and this is where *probabilistic computation* enters. Probabilistic machines are effectively an intermediary: they are deterministic machines that simulate nondeterministic choices by way of consulting random values. Although a probabilistic machine can't evaluate every computation branch at once in the way that a nondeterministic machine can, it can still "make guesses" by following one of potentially many computation branches with some prespecified odds.

Of course, this presents a downside: if the "guesses" made by a probabilistic machine are incorrect, this may lead to the machine producing an incorrect output, say by mistakenly rejecting a word that does in fact belong to the language of the machine. This naturally isn't an issue with nondeterministic machines, since all computation branches are evaluated at once and we only require the existence of one accepting computation branch to produce a positive outcome. But despite the risk of incorrect outputs, probabilistic machines do possess remarkable utility: by allowing the machine to produce incorrect outputs with a small probability, we are often able to solve problems in less time or space than would be used by a deterministic machine.

1.1 Definition

In order for us to perform probabilistic computations, we must define a model of computation that is capable of using randomness. This model of computation, a *probabilistic Turing machine*, is fundamentally the same as any other Turing machine we've seen thus far, but its transition "function" is split into two distinct functions δ_1 and δ_2 that the machine applies with equal probability. That is, the machine applies δ_1 with probability $1/2$ or otherwise it applies δ_2, akin to flipping a fair coin.

Definition 1 (Probabilistic Turing machine)**.** A probabilistic Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta_1, \delta_2, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet* (where $\sqcup \notin \Sigma$);

- Γ is the *tape alphabet* (where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$);
- $\delta_1 : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *first transition function*;
- $\delta_2 : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *second transition function*;
- $q_0 \in Q$ is the *initial* or *start state*;
- $q_{\text{accept}} \in Q$ is the *final* or *accepting state*; and
- $q_{\text{reject}} \in Q$ is the *rejecting state*.

Note that δ_1 and δ_2 are fundamentally the same function in that they each map a tuple of state and tape symbol to a tuple of state, tape symbol, and input head movement, but they do not necessarily have the same behaviour. As an illustrative example, supposing we are in some state q_i and reading some symbol $\mathtt{a}$, δ_1 may produce the tuple $(q_j, \mathtt{b}, R)$ while δ_2 may produce the tuple $(q_k, \mathtt{b}, L)$. In essence, we have two choices for each transition, and each choice occurs with probability 1/2.

Remark. We may alternatively define a probabilistic Turing machine to be a deterministic Turing machine that has access to a special read-only *random tape* filled with random bits. Under this definition, the machine can choose which of its transition functions to apply by consulting the bits on the random tape.

1.2 Computations and Accepting Computations

While a deterministic computation either always accepts or always rejects its input word, and a nondeterministic computation accepts if there exists at least one accepting computation branch, probabilistic computations behave in a somewhat different manner. To each input word, a probabilistic Turing machine associates a value between 0 and 1 that corresponds to the probability that a randomly selected computation branch accepts that word. Thus, some percentage of computation branches will accept the word, while the other branches will reject it.

Since our definition specifies that a probabilistic Turing machine applies either transition function with probability 1/2, every step of the machine's computation can proceed in one of two ways, and so the computation tree resembles a binary tree. For deterministic portions of the computation, both branches will lead to the same configuration, while for nondeterministic portions of the computation having more than two possible outcomes, the structure of the computation can be reorganized to produce exactly two branches at each step.

In this way, for any probabilistic Turing machine $\mathcal{M}$, we can assign to each branch b of its computation tree the probability

$$\mathbb{P}[b] = \frac{1}{2^k},$$

where k is the number of computation steps that occurred along the branch b. As an immediate consequence, the probability that $\mathcal{M}$ accepts an input word w is

$$\mathbb{P}[\mathcal{M} \text{ accepts } w] = \sum_{b \in \text{ACC}} \mathbb{P}[b],$$

where ACC is the set of all halting computations on w that lead to an accepting state of $\mathcal{M}$. At the same time, the probability that $\mathcal{M}$ rejects w is simply

$$\mathbb{P}[\mathcal{M} \text{ rejects } w] = 1 - \mathbb{P}[\mathcal{M} \text{ accepts } w].$$

Lastly, the language of a probabilistic Turing machine is the set of all words whose probability of acceptance is above some predefined threshold. There are many ways we can define such a threshold, and these myriad ways form the cornerstone of the study of randomized complexity theory.

1.3 Probabilistic Resource Bounds

When it comes to measuring the resource usage of a probabilistic Turing machine, things aren't quite as straightforward as when we were dealing with deterministic or nondeterministic Turing machines. Since a probabilistic Turing machine chooses computation branches at random, we have no way of knowing in advance whether we might follow a branch that halts quickly or a branch that takes ages to complete. Thus, before we can define time and space bounds for probabilistic Turing machines, we must come up with some way to accurately measure the resources used by an individual computation branch that is chosen at random!

To make our job easier, we will introduce the following assumption on the structure of the computation tree of any probabilistic Turing machine computing a length-n input word in time at most $t(n)$.

ABEL property. Let $\mathcal{M}$ be a $t(n)$-time probabilistic Turing machine. The computation of $\mathcal{M}$ on a length-n input word and with any sequence of random bits (or "coin tosses") halts in exactly $t(n)$ steps.

The acronym ABEL stands for "all branches, equal length"; under this assumption, every branch of the computation tree has the same length, which makes our job of analyzing the computation much easier.

If we suppose the computation of some probabilistic Turing machine $\mathcal{M}$ satisfies the ABEL property, then since every computation step of $\mathcal{M}$ can lead to one of two choices and since every computation of $\mathcal{M}$ takes time $t(|w|)$ by the ABEL property, the computation tree of $\mathcal{M}$ consists of exactly $2^{t(|w|)}$ branches. For any input word w, the probability that $\mathcal{M}$ accepts w is given by the expression

$$\mathbb{P}[\mathcal{M} \text{ accepts } w] = \frac{|\text{ACC}|}{2^{t(|w|)}},$$

where ACC again denotes the set of all halting computations on w that lead to an accepting state of $\mathcal{M}$.

Thankfully, the ABEL property is not so strong that we lose generality by taking it as an assumption. As long as $t(n)$ is a time-constructible function, we can take any arbitrary $t(n)$-time probabilistic Turing machine and construct an equivalent probabilistic Turing machine that satisfies the ABEL property.

2 Las Vegas and Monte Carlo Algorithms

Due to their inherent computational behaviour, probabilistic Turing machines run *randomized algorithms*. However, when it comes to an algorithm, not all randomized behaviour is the same: some algorithms differ in how many resources are used over the course of the computation, while others differ in the probability that the algorithm outputs a correct answer.

We can classify all randomized algorithms as one of two types (well, strictly speaking, three types), depending on whether the randomization aspect applies to the runtime of the algorithm or to the output produced by the algorithm.

- A *Las Vegas algorithm* always gives us a correct output, but the runtime of the algorithm may vary depending on the random choices made by the algorithm.
- A *Monte Carlo algorithm* is guaranteed to run in a certain amount of time, but the output produced may be subject to error depending on the random choices made by the algorithm. Specifically:
  - A Monte Carlo algorithm *with one-sided error* is always correct when it returns one answer ("yes") and is incorrect with some bounded probability when it returns the other answer ("no"), or vice versa.
  - A Monte Carlo algorithm *with two-sided error* is incorrect with some bounded probability regardless of the answer it returns ("yes" or "no").

As a small example, suppose we have an array A containing n elements: $(n-1)$ zeroes and 1 one. We can design two randomized algorithms to find the index value i such that $A[i] = 1$:

Algorithm 1: Array search—Las Vegas

```
while true do
    i ← random integer
    if A[i] = 1 then
        return i
```

Algorithm 2: Array search—Monte Carlo

```
for 0 ≤ c ≤ 10 do
    i ← random integer
    if A[i] = 1 then
        return i
return failure
```

With the Las Vegas algorithm, we're guaranteed to find the index value, but we could potentially loop for a very long time if we keep making unlucky random choices. With the Monte Carlo algorithm, we only loop 10 times; we will always return the index value if it is found, but we may instead return "failure" if we don't find the index value (despite the fact that it exists). Therefore, this particular Monte Carlo algorithm has one-sided error.

The relationship between Las Vegas and Monte Carlo algorithms can be summarized in the following table:

	Correctness	**Runtime**
Las Vegas	Certain	Uncertain
Monte Carlo	Uncertain	Certain

Alternatively, if you prefer a mnemonic to distinguish between the two, remember that **L**as **V**egas algorithms always **V**alidate their output but sometimes **L**ose track of time, while **M**onte **C**arlo algorithms always **C**ontrol their time but sometimes **M**istake their output.

Having distinguished between the two types of randomized algorithms, a natural question to ask might be "can we convert between the two?" That is, if we have a randomized algorithm of one type, can we adapt the way it applies randomization to be of the other type? The answer is... sometimes.

If we start with a Las Vegas algorithm, it's rather straightforward for us to adapt the algorithm to run in a fixed amount of time at the expense of introducing error into the output we get.

Theorem 2. *Every Las Vegas algorithm can be converted to a Monte Carlo algorithm.*

Proof. Omitted. □

On the other hand, we cannot in general convert a Monte Carlo algorithm to a Las Vegas algorithm unless we have some method of testing the correctness of the output produced by the algorithm. If we have such a testing method, then we can simply repeat the execution of the Monte Carlo algorithm until we get a correct output. Otherwise, we must rely on whatever knowledge we have of the distribution of outputs in order to determine our level of confidence in the output given by a Monte Carlo algorithm.

3 Randomized Time Complexity

As with our deterministic and nondeterministic algorithms, there is a rich tapestry of complexity classes that characterize the behaviour of randomized algorithms under different conditions. Depending on the type of randomized algorithm we're taking under consideration, and on the particular threshold we fix on the probability of acceptance, we obtain one of many randomized complexity classes. Each of these complexity classes also interacts in interesting ways with our fundamental classes like P, NP, and the rest. In this section, we consider the major randomized complexity classes in turn.

3.1 Two-Sided Error: PP and BPP

We will begin our study of randomized complexity theory by devising a probabilistic analogue of the class P; that is, a class comprised of decision problems that can be solved by a probabilistic Turing machine

in some polynomial amount of time $p(n)$, independent of any random choices made over the course of the computation. Supposing that $p(n)$ is time constructible, it's easy for the probabilistic Turing machine to abide by this time bound: it simply counts the number of computation steps it performs and halts once the limit has been reached.

The simplest way we can define a randomized complexity class is just to fix some threshold value and state that an input word will be accepted by the probabilistic Turing machine if its probability of acceptance is greater than this threshold.

Since the problems we are currently considering can be solved by a polynomial-time probabilistic Turing machine, the name of our first complexity class will be PP, representing all *probabilistic polynomial time* decision problems.

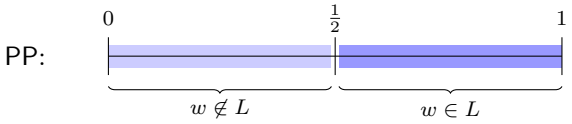
Definition 3 (The class PP)**.** A decision problem L belongs to the complexity class PP if there exists a probabilistic Turing machine $\mathcal{M}$ such that, given an input word w,

- if $w \in L$, then $\mathbb{P}[\mathcal{M} \text{ accepts } w] > 1/2$; and
- if $w \notin L$, then $\mathbb{P}[\mathcal{M} \text{ accepts } w] < 1/2$.

Note that the class PP corresponds to the class of decision problems that can be solved by a Monte Carlo algorithm with two-sided error. If $w \in L$, then a PP-machine will reject w with probability less than $1/2$, while if $w \notin L$, then the machine will accept w with probability less than $1/2$.

For those who are curious why we selected $1/2$ as our threshold, there's no particular reason why we selected that value specifically. Indeed, we could choose any value $0 < \alpha < 1$ as our threshold, and the definition of PP would follow in exactly the same way for α as it did for $1/2$. We simply went with $1/2$ as it's a "nice" value to work with.

Now, with randomized complexity classes, it's often helpful to illustrate on a probability line where exactly the accepting and rejecting probabilities may fall. Probability lines are a great tool to compare and contrast randomized complexity classes, since they allow us to differentiate classes at a glance. If we were to draw a "picture" of the class PP on a probability line and highlight the regions of this line where the probability of acceptance would appear given either $w \in L$ or $w \notin L$, we would have the following:



Unfortunately, as you may have noticed in our illustration, one consequence of our definition of the class PP is that there exists a tricky *probability gap* between the probability of a PP-machine accepting inputs $w \in L$ and the probability of the same machine accepting inputs $w \notin L$. Since the only condition is that we accept words $w \in L$ with probability strictly greater than $1/2$, it is possible to squeeze the acceptance probability to be arbitrarily close to $1/2$ and make it increasingly difficult for us to distinguish between correct and incorrect outputs.

For example, it is possible for us to design a PP-machine that accepts inputs $w \in L$ with probability $1/2+1/2^n$ and accepts inputs $w \notin L$ with probability $1/2-1/2^n$, where $n = |w|$. As n grows, the gap between correctly accepting a word in L and incorrectly accepting a word not in L becomes arbitrarily small, and we must devote increasing attention to making sure our outputs are correct.

While the existence of an arbitrarily small probability gap alone suggests that working with the class PP can be difficult, we can go one step further by showing that PP is such a powerful class that it contains problems that are not known to be efficiently computable!

Recalling the class NP, we can define this class probabilistically as the class of all decision problems L for which there exists a probabilistic Turing machine $\mathcal{M}$ where, for all $w \in L$, $\mathbb{P}[\mathcal{M} \text{ accepts } w] > 0$. This aligns with the definition of a nondeterministic computation, where we accept an input word if there exists at least one accepting path in the computation tree.