CS 252 - Probabilistic Turing Machines (Algorithms)
Additional Reading 3
and
Homework problems

3 Probabilistic Turing Machines

When we talked about computability, we found that nondeterminism was sometimes a convenient mechanism for designing algorithms but that it could be simulated deterministically; that is, we found that "exotic" nondeterministic algorithms have no greater *computational* ability than do their mundane deterministic cousins. However, now that we have begun to consider issues of complexity, we are seeing evidence that nondeterminism may have the advantage in terms of efficiency. Unfortunately, since we cannot actually implement nondeterministic algorithms, they serve only as a mathematical foil for reasoning about complexity classes.

The idea of considering many different computational paths is appealing, and though we cannot do so nondeterministically, *probabilistic Turing machines* are a deterministic "approximation" that we can implement and that can be useful in some situations in which vanilla deterministic Turing machines may not be. The high-level idea is that each nondeterministic choice is simulated probabilistically. Different choices have different probabilities, and the probability of a single computational history is the product of the probabilities of the choices used to follow that history. Then, running such a machine can be thought of as effectively probabilistically choosing one of the many nondeterministic "universes" to sample.

While this is computational feasible, it introduces a difficulty—the probabilistic path chosen may result in an incorrect answer. This is not an issue for a nondeterministic machine because it "magically" tries all possible computational paths and if any of them accept, it accepts. However, since we are now probabilistically sampling a single path, it may end in the reject state, even though some other path would accept. So, the question is, if we allow a Turing machine to be wrong sometimes, can it still be useful? The answer is yes, under certain reasonable conditions.

3.1 Formal Definition of Probabilistic Turing Machines

The formal definition of probabilistic Turing machine shares commonalities with DFAs and Markov chains.

Definition 0.0.1. A ***Probabilistic Turing Machine*** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, \gamma, q_0, q_{accept}, q_{reject})$, where

1. Q is a finite set called the ***states***
2. Σ is a finite set called the ***input alphabet*** that does not contain the blank symbol $\square$
3. Γ is a finite set called the ***tape alphabet***, where $\square \in \Gamma$ and $\Sigma \subseteq \Gamma$

4. $\delta : Q \times \Gamma \times Q \cup \{q_{accept}\} \cup \{q_{reject}\} \times \Gamma \times \{L, R\} \rightarrow [0 \ldots 1]$ is the ***transition function***

5. q_0 is the ***start state***

6. $q_{accept} \notin Q$ is the ***accept state***

7. $q_{reject} \notin Q$ is the ***reject state***

Note that this only differs from the definition of a deterministic Turing machine in the transition function δ. Here, like for our treatment of Markov chains, we define δ to return a number in the range $[0 \ldots 1]$, which we interpret as a probability.

3.2 Computing with Probabilistic Turing Machines

For a probabilistic TM P, we take a similar approach to that of a deterministic TM, generalizing the idea as follows. We now say that a configuration C_1 *probabilistically yields* a configuration C_2 with probability p if the probabilistic transition function maps the two configurations to p.[1] Let $T = \{T_1, T_2, \ldots, T_n\}$ be the set of all possible computational *paths* of P on input w and let a path T_i consist of a sequence of configurations $C_{i,1}, C_{i,2}, \ldots, C_{i,k_i}$. We say that path T_i accepts w with probability p_{T_i} if

1. $C_{i,1}$ is the start configuration of P on input w

2. each $C_{i,j}$ probabilistically yields $C_{i,j+1}$ with probability p_j

3. C_{i,k_i} is an accepting configuration

4. $p_{T_i} = \Pi_j p_j$

Let $A \subseteq T$ be the set of accepting paths of P. Then, we say that probabilistic TM P accepts string w with probability $p(w) = \sum_{C \in A} p_C$.

3.3 A simple example

Figure 1 visualizes the computational paths of a simple probabilistic Turing machine, with each branch having an associated probability (given by the transition function δ) and with some paths being accepting (the set A contains those paths numbered 1, 3 and 7) and others rejecting (those numbered 2, 4, 5, 6, 8; recall that we have restricted ourselves to decidable problems, so all paths are of finite length). The starting configuration C_1 represents the machine in its start state reading the first symbol of the input string w. The configuration C_2 represents one possible configuration after one computational step, and that computational step will be chosen with probability 0.7. With probability 0.3, the other possible computational choice from the initial configuration will be chosen and the result will be configuration

[1]More formally, let $a_1, a_2, b \in \Sigma$, $y, z \in \Sigma^*$. Then, C_1 probabilistically yields C_2 with probability p if $C_1 = yq_1a_1a_2z$, $C_2 = ybq_2a_2z$ and $\delta(q_1, a_1, q_2, b, R) = p$ or if $C_1 = ya_2q_1a_1z$, $C_2 = yq_2a_2bz$ and $\delta(q_1, a_1, q_2, b, L) = p$.

C_3, and so on. The probability that path 1 accepts is $p_1 = 0.7 \times 0.5 \times 0.8 = 0.28$. The probability that the machine accepts the string w is the sum of the probabilities of the accepting paths, $p(w) = p_1 + p_3 + p_7 = 0.28 + 0.21 + 0.063 = 0.553$.

3.4 The Class BPP

The complexity class BPP is the probabilistic counterpart to the complexity class P. It contains all languages that can be decided in polynomial time with a probabilistic TM whose error bound ϵ is better than random guessing (so, because all languages in P can be decided with 0 error, in particular P $\subseteq$ BPP). The trick is guaranteeing a bound on ϵ, and that is language dependent and beyond the scope of this treatment.

Definition 0.0.2. BPP is the class of languages that can be decided by a probabilistic polynomial time Turing machine with an error probability $\epsilon < \frac{1}{2}$.

Note that the error bound must be strictly less than 0.5, but it can be arbitrarily close to 0.5. That is to say, as long as the machine is doing slightly better than random guessing, it will be good enough. While this seems like an unsatisfactory kind of an algorithm (especially in safety-critical situations), given such a "weak" solution, the following amplification technique allows us to use it to construct a "strong" solution, one that can have an arbitrarily low error probability while running in a reasonable amount of time.

3.4.1 Amplification

Amplification is a very simple idea—run the weak algorithm multiple times and take the majority answer. The question, of course, is how many times the weak algorithm must be run before we are confident that the majority answer represents the correct answer.

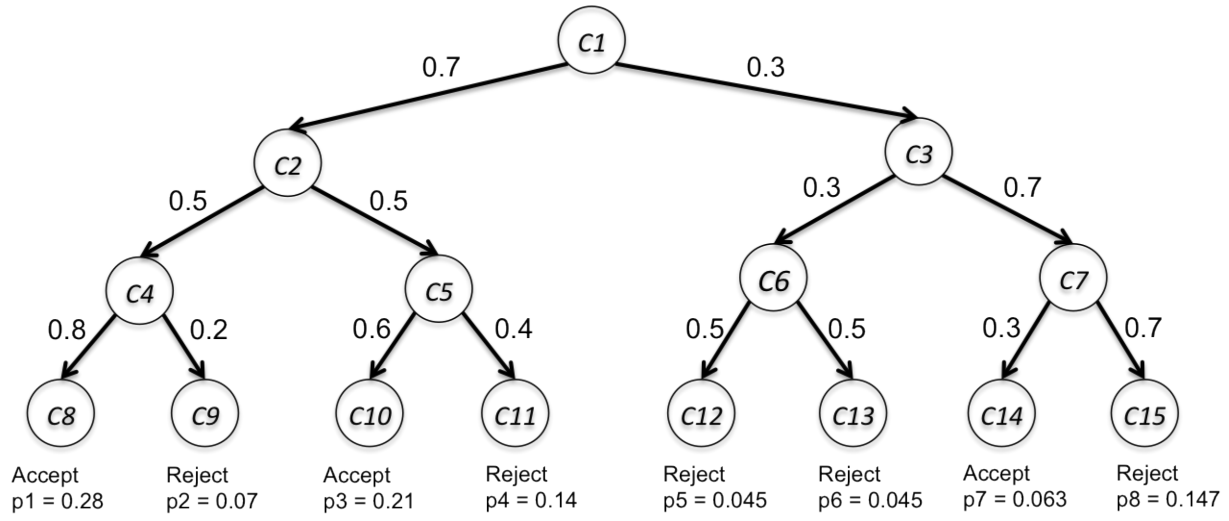

Figure 1: Eight probabilistic computational paths for a simple probabilistic TM.

Fortunately, we don't have to run it very many times before we can start to be confident. Of course, the weaker the algorithm (the closer its error is to 0.5), the more times we will have to run it; and, the more confident we want to be, the more times we'll have to run it. But, remarkably, the number of runs needed scales only polynomially with the error rate and the desired confidence.

Suppose we have a weak probabilistic TM P_1 with error rate $\epsilon_1 < 0.5$. We create a strong probabilistic TM P_2, with error rate ϵ_2 as follows:

```
$P_2$ = on input $< w, \beta >$
1. Calculate $k$ for desired error bound $\beta$
2. Run $2k$ simulations of $P_1$ on input $w$
3. If most runs of $P_1$ accept, then *accept*; otherwise, *reject*
```

Since P_2 outputs the majority decision of multiple runs of P_1, it will answer incorrectly only when the majority of P_1's outputs are incorrect. That is, ϵ_2 is the probability that $2k$ runs of P_1 produce a misleading majority. What is this probability? While finding an exact answer might be difficult, we can find an upper bound on the probability. Indeed, we will claim that it drops exponentially with each simulation of P_1 and that, as a result, we can bound the error as low as we'd like with a polynomial number of runs of P_1. We offer here without proof, the following claims (see Appendix for proofs):

Claim 1: *Error rate $\epsilon_2 \leq \beta$.*

Claim 2: *The runtime of P_2 is polynomial in $|w|$ and β.*

The result of these claims is that we can drive the error rate of P_2 very low very quickly, quickly enough that we can become arbitrarily confident in the answer, even if it is based on a very weak P_1.

3.4.2 An example

Suppose P_1 has an error rate of $\epsilon_1 = 0.3$, and suppose we would like to use it to build P_2 such that $\epsilon_2 \leq 0.001$. If we choose $k = 40$ and therefore run P_1 80 times and report the majority decision (see Appendix for how to calculate k), then by Claim 3 (see Appendix) the error probability ϵ_2 for this majority vote is bounded by

$$\epsilon_2 \leq (4\epsilon_1(1-\epsilon_1))^k \leq (4(0.3)(0.7))^{40} = 0.000935775008617 \leq 0.001$$

3.5 Exercises

Exercise 3.1. Consider the following scenario. A high school senior is taking a college entrance exam to determine which of two universities she will attend. The exam consists of a single True/False question. If the student answers correctly, she will attend Blue University. If the student answers incorrectly, she will attend The University of Red. Because the question is so difficult, the student is allowed access to a probabilistic TM that gives the correct answer only 50.001% of the time. The student does not know the answer and knows she must guess, but she would like to be 99.99999% confident in her guess. How many times should she query the TM before she chooses her answer?

Exercise 3.2. Suppose we have a language A, a polynomial time Turing machine M and two fixed error bounds ϵ_1 and ϵ_2 where$0 < \epsilon_1 < \epsilon_2 < 1$. Further suppose that M works as follows: a. For any $w \in A$, M accepts with probability at least ϵ_2. b. For any $w \notin A$, M accepts with probability no greater than ϵ_1.

Think about the result of using an amplification technique on M and explain why $A \in BPP$ (even though we have two error bounds, one or both of which might be greater than $\frac{1}{2}$).

3.6 Appendix. Math: Proceed with Caution

Claim 1: *Error rate $\epsilon_2 \leq \beta$.*

Proof. Let

$$k = \left\lceil \frac{\log_2(\frac{1}{\beta})}{-\log_2(4\epsilon_1(1-\epsilon_1))} \right\rceil$$

Then, using logarithmic identities, we have

$$k \geq \frac{\log_2(\frac{1}{\beta})}{-\log_2(4\epsilon_1(1-\epsilon_1))} = -\log_{(4\epsilon_1(1-\epsilon_1))}(\frac{1}{\beta}) = \log_{(4\epsilon_1(1-\epsilon_1))}(\beta)$$

Since $\epsilon_1 < 0.5$ and therefore $(4\epsilon_1(1-\epsilon_1)) < 1$, by Claim 3 below,

$$\epsilon_2 \leq (4\epsilon_1(1-\epsilon_1))^k \leq (4\epsilon_1(1-\epsilon_1))^{\log_{(4\epsilon_1(1-\epsilon_1))}(\beta)} = \beta$$

□

Claim 2: *The runtime of P_2 is polynomial in $|w|$ and β.*

Proof. P_2 is composed of three steps. Step 1 is a simple calculation for the value of k (given in the proof Claim 1) and is clearly $O(\text{poly}(|w|, \beta))$. Step 2 is a loop that runs $2k$ times. Each pass through the loop consists of simulating P_1, which by assumption is polynomial $O(\text{poly}(|w|))$. Since the value of $k \approx \log \beta$, it is clearly $O(\text{poly}(\beta))$. Combining, we have that step 2 is $O(\text{poly}(\beta) * \text{poly}(|w|))$. Step 3 is $O(1)$. So, the entire algorithm is $O(\text{poly}(|w|, \beta)) + O(\text{poly}(\beta) * \text{poly}(|w|)) + O(1)$. □

Claim 3: *Error rate $\epsilon_2 \leq (4\epsilon_1(1-\epsilon_1))^k$.*

Proof. Recall that $\epsilon_1 < 0.5$ is the probability that P_1 outputs an incorrect answer. Then, $(1-\epsilon_1) > 0.5$ is the probability that P_1 outputs correctly. Given a sequence s of $2k$ outputs from P_1, let c be the number of correct answers and d be the number of incorrect answers in the sequence. Call a sequence *misleading* if $d \geq c$ because when $d \geq c$, P_2 is misled by the majority and outputs incorrectly. Let S_m be the set of all misleading sequences. Then the error probability ϵ_2 for P_2 can informally be expressed as

$\epsilon_2 =$ the probability that $2k$ runs of P_1 will produce a misleading sequence $s \in S_m$

Letting p_s represent the probability of a single simulation of P_1 producing sequence s, we can say more formally,

$$\epsilon_2 = \sum_{s \in S_m} p_s$$

Since the simulation runs are independent, the probability of seeing a particular sequence s with c correct answers and d incorrect answers is

$$p_s = \epsilon_1^d (1 - \epsilon_1)^c$$

For a misleading sequence $s \in S_m$, since $d + c = 2k$ and $d \geq c$ and $(1 - \epsilon_1) > \epsilon_1$,

$$p_s = \epsilon_1^d (1 - \epsilon_1)^c \leq \epsilon_1^k (1 - \epsilon_1)^k$$

Finally, since the total number all possible sequences is 2^{2k} and therefore $|S_m| \leq 2^{2k}$, we can say

$$\epsilon_2 = \sum_{s \in S_m} p_s \leq \sum_{s \in S_m} \epsilon_1^k (1 - \epsilon_1)^k \leq 2^{2k} \epsilon_1^k (1 - \epsilon_1)^k = (4\epsilon_1(1 - \epsilon_1))^k$$

$\square$