6.842 Randomness and Computation February 14, 2022

Lecture 5

Lecturer: Ronitt Rubinfeld *Scribe: Nadia Koshima*

1 Overview

- Uniform Generation
  - Example with uniform generation of DNF solutions
- Counting Problems
  - Counting classes
  - #DNF
- Approximate Counting
  - Counting classes, #P
  - Exact vs Approximate Counting
  - Approximate #DNF
  - Downward Self-Reducibility Trees

2 Uniform Generation: DNF Formula

Uniform generation is a type of problem where we aim to create a uniform distribution from one that is variable. As an example, we will be looking at an algorithm for randomly sampling solutions for a DNF formula.

2.1 Disjunctive Normal Form(DNF)

DNF Formulas are boolean formulas that can be described as "**OR of ANDs**." DNF formulas take the general form of

$$\varphi(x_1, x_2, ..., x_n) = C_1 \vee C_2 \vee ... \vee C_m$$

where conjunction $C_i = x_{i_1} \wedge x_{i_2} \wedge ... \wedge x_{i_{|C_i|}}$ and $x_{i_j} \in \{x_1, \bar{x_1}, x_2, \bar{x_2}, ...\}$

An example of this would be

$$\varphi(x_1, ..., x_n) = x_1\bar{x_2}x_3 \vee x_2\bar{x_3}x_4x_{10} \vee x_8\bar{x_{10}}x_{11} \vee ...$$

Note how we can write multiplication of variables as implicit ANDs ($\wedge$), and generally can leave them out of notation.

2.2 Task: Find one satisfying assignment to φ

This is pretty easy! We just have to pick one literal and set the respective variables to TRUE (or false if $\bar{x_i}$), and then set all other variables randomly. The only case in which there is no satisfying assignment is if x_i and $\bar{x_i}$ both exist in that literal because then we have a contradiction.

2.3 Task: Find random satisfying assignment to φ

This is a bit trickier since, by random, we mean that we want to be able to sample a satisfying assignment uniformly.

2.3.1 Special Case: only one conjunction

In this case we only have one conjunction:

$$\varphi(x_1, x_2, ..., x_n) = x_{i_1} x_{i_2} ... x_{i_{|C_1|}} \text{ for } x_{i_j} \in \{x_1, \bar{x_1}, x_2, \bar{x_2}, ...\}$$

For example:

$$F(x_1, x_2, x_3, ..., x_n) = x_1 \bar{x_2} x_3$$

In this example, the satisfying assignment would be any s.t.

$$x_1 = T \ , \ x_2 = F \ , \ x_3 = T$$

Therefore, to generate a random satisfying assignment to F:
Let $x_1 = T \ , \ x_2 = F \ , \ x_3 = T$
and pick $x_4, ..., x_n$ randomly from $\{T, F\}$

In general, in the case of one conjunction, we can satisfy the literal and set other variables randomly.

2.3.2 Two Conjunction Case

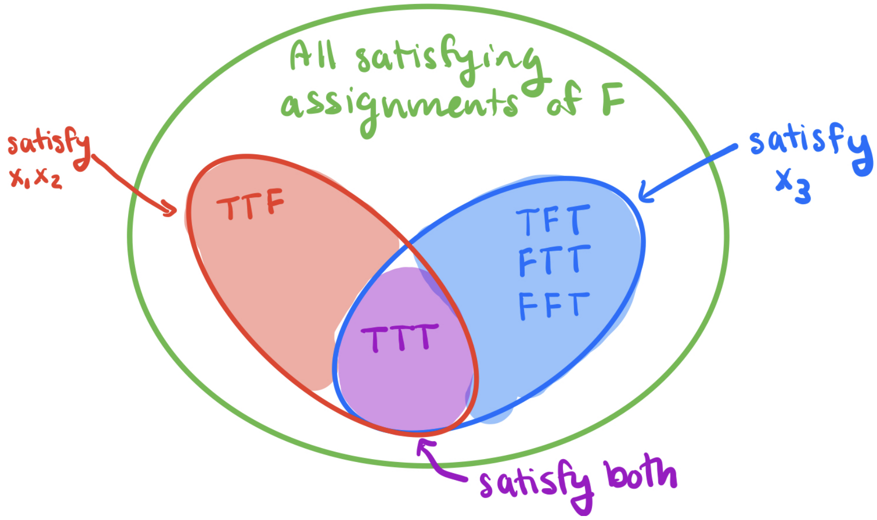
Our algorithm attempt will be as follows:

1. Pick $i \in \{1, 2\}$ (essentially pick one of the two conjunctions to satisfy)
2. Set variables in conjunction C_i to TRUE
3. Set other variables randomly

Example: $F = x_1 x_2 \vee x_3$

1. Pick conjunction 1
2. Set $x_1 = x_2 = T$
3. Set $x_3 = T$

Is this a uniform sampling? We can look at the space of satisfying assignments, grouped by conjectures:



From the image, we can see that the assignment TTT has a higher probability of being selected, as it is a satisfying assignment to both conjunctions. Furthermore, we can see the number of satisfying assignments for each conjunction also creates an unequal distribution, as x_1x_2 has less assignments. Both issues create unfair sampling. To look at the probability of sampling each assignment more specifically:

$$P(\text{randomly sampling } \textbf{TTF} \text{ in F}) = P(\text{randomly sampling } C_1) * P(\text{randomly sampling TTF in } C_1)$$

$$= \frac{1}{2}\frac{1}{2}$$

$$= \frac{1}{4}$$

$$P(\text{randomly sampling } \textbf{TFT} \text{ in F}) = \frac{1}{2}\frac{1}{4} = \frac{1}{8}$$

$$P(\text{randomly sampling } \textbf{FTT} \text{ in F}) = \frac{1}{8}$$

$$P(\text{randomly sampling } \textbf{FFT} \text{ in F}) = \frac{1}{8}$$

$$P(\text{randomly sampling } \textbf{TTT} \text{ in F}) = P(\text{randomly sampling } C_1) * P(\text{randomly sampling TTF in } C_1)$$

$$+P(\text{randomly sampling } C_2) * P(\text{randomly sampling TTF in } C_2)$$

$$= \frac{1}{4} + \frac{1}{8} = \frac{3}{8}$$

To summarize why this variable probability distribution occurs in sampling F:

1. Second conjunction has more assignments which means $P(\text{randomly sampling S in } C_j)$ where S is one satisfying assignment for C_j is not uniform over all j

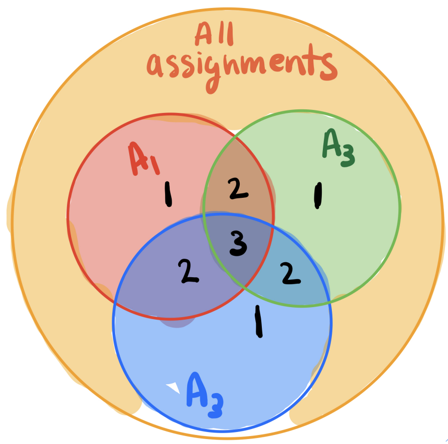
2. Some assignments may satisfy more than one conjunction (i.e. TTT in this case)

2.3.3 Fixed algorithm for any number of conjunctions

To fix the problems with the previous algorithm, we will use the two following fixes:

1. Sample conjunction **proportionally to the number of satisfying assignments** (ie for C_i proportional to $|C_i|$)
2. If assignment satisfies more than one conjunction, "correct" for it by using **rejection sampling**

One way we might view the space of satisfying conjunctions for a DNF formula with three conjunctions is:



Here, the numbers in black show how many conjunctions an assignment occupying that space would satisfy and is somewhat proportional to how much more likely that assignment would be sampled, using our initial naive method. For example, an assignment at the intersection of all three conjunctions would be approximately three times more likely to be sampled – to correct this, we toss a coin with bias of $\frac{1}{3}$ to decide whether or not to accept, thus performing rejection sampling. We would also need to do this for an assignment at the intersection of any two different conjuctions – again, we want to correct with a coin toss (this time with bias $\frac{1}{2}$).
Now that we understand the intuition behind what changes we might make, we can specify the algorithm.

Algorithm: Let A_i be the set of assignments that satisfy conjunction C_i, specifically

$$A_i \leftarrow \{\bar{x} = (x_1, x_2, ..., x_n) | \bar{x} \text{ satisfies } C_i\}$$

Algorithm 1 Algorithm for randomly sampling a satisfying assignment

---

input $\varphi = \vee_{i=1}^{m} C_i$ (formula)
Let $A_i \leftarrow \{\bar{x} = (x_1, x_2, ..., x_n) | \bar{x} \text{ satisfies } C_i\}$
repeat
Pick i with probability $\frac{|A_i|}{\sum_{j=1}^{m} |A_j|}$ ▷ $|A_i| = 2^{n-k}$where k is # of variables in A_i
Pick uniform assignment $\bar{b}$ in A_i
Let $t_{\bar{b}} \leftarrow |\{j | \bar{b} \text{ satisfies } A_j\}|$ ▷ $t_{\bar{b}}$ is the number of conjunctions satisfied by $\bar{b}$
Output $\bar{b}$ with probability $\frac{1}{b}$
until we succeed with an accepted assignment

Note: In each round, $t_b \geq 1$ since we know that it at least satisfies A_i.

How this algorithm generates uniformity: We will look at the probability that some satisfying assignment $\bar{b}$ is output in any round.

$$P(\text{output } \bar{b} \text{ in round } i) = P(\text{accepting } \bar{b}) \left(\sum_{j \in [m] | \bar{b} \in A_j} P(\text{pick j in round i}) * P(\text{pick } \bar{b} \text{ from } A_j) \right)$$

$$P(\text{output } \bar{b} \text{ in round } i) = \frac{1}{t_{\bar{b}}} \left(\sum_{j \in [m] | \bar{b} \in A_j} \frac{|A_j|}{\sum_{k=1}^{m} |A_k|} * \frac{1}{|A_j|} \right)$$

Note that $|A_i|$ cancels out in the summation, and as a result, the summation now contains a probability independent of j. Therefore, we get:

$$P(\text{output } \bar{b} \text{ in round } i) = \frac{1}{t_{\bar{b}}} * \frac{t_{\bar{b}}}{\sum_{k=1}^{m} |A_k|}$$

$$P(\text{output } \bar{b} \text{ in round } i) = \frac{1}{\sum_{k=1}^{m} |A_k|}$$

Note that the resulting probability is the same for all $\bar{b}$ regardless of how many conjunctions it satisfies, and thus, we we have achieved uniform sampling.

Runtime: We will now look at the runtime of this algorithm.

$$P(\text{loop succeeds} \geq \frac{1}{\max t_{\bar{b}}} \geq \frac{1}{m}$$

Therefore,

$$E[\# \text{ of loops until succeed}] \geq m$$

The runtime of each loop is poly(m+n).

3 Counting Problems

3.1 Counting Classes

Counting complexity classes involve being able to *count* the number of accepted solutions to a given problem (SAT, 3SAT, Knapsack, Subset sum, etc) in a certain amount of time (P, NP, EXP, etc).

Definition 1 (#P) Class of problems that count the number of accepted paths in poly-time non-deterministic Turing Machines.

Definition 2 (#P-Complete) Class of problems that are

- in #P
- every problem in #P has a Turing reduction or poly-time reduction to it

The *#SAT* problem, which counts the number of satisfying assignments to Boolean formula φ, is *#P-Complete.*

3.2 #DNF

We might think that, since DNF is in P, that this problem would be easier. One initial thought to approaching this problem is counting the number of solutions of each A_i, without taking intersections into consideration. Then, we could use Inclusion-Exclusion Principle to account for assignments that are repeated. However, the problem with this is that it would create an exponential amount of terms, meaning that our runtime would also be exponential.
It turns out that #DNF is computationally hard. One reason why is due to this fact: *if $\#DNF \in P$, then $\#CNF \in P$*. Why? Well, given any φ in CNF:

$$\varphi \text{ is satisfiable iff } \bar{\varphi} \text{ has } > \text{ one unsatisfying assignment (or } < 2^n - 1 \text{ satisfying assigenemnts)}$$

Or a better way to view it:

$$\# \text{ of assignments to } \varphi = 2^n - \# \text{ assignments to } \bar{\varphi}$$

Why is this true? Well, we can use DeMorgan's Law to convert between C$\bar{\text{N}}$F formulas and DNF formulas:

Definition 3 (DeMorgan's Law) Shows how to relate Boolean statements to their opposites:

- $\overline{A \vee B} = \overline{A} \wedge \overline{B}$
- $\overline{A \wedge B} = \overline{A} \vee \overline{B}$

Notice that the second equation shows how to convert a CNF formula ($\bar{\varphi}$) to a DNF formula. Therefore, we can reduce counting the number of assignments that CNF formula φ has to counting the number of assignments DNF formula $\bar{\varphi}$ has, which is in #DNF.
However, it turns out that #DNF is #P-complete.

4 Approximate Counting

Definition 4 (Randomized Approximation Scheme) Given Boolean formula φ and $\epsilon > 0$ as a parameter. Define z to be the number of satisfying assignments to φ. A *randomized approximation scheme* is one that outputs y such that

$$\frac{z}{1+\epsilon} \leq y \leq z(1+\epsilon)$$

with probability $\geq \frac{3}{4}$.

Definition 5 (Fully Polynomial Randomized Approximation Scheme (FPRAs)) *FPRAs* are randomized approximation schemes (as defined above) that are polynomial in $|\varphi|$ and $\frac{1}{\epsilon}$.

4.1 FPRAs for SAT → randomized PTime algorithm for SAT

In this section, we will prove how having an FRPA for SAT implies that there exists a randomized PTime algorithm for SAT.

Algorithm 2 Algorithm for converting SAT FPRA to SAT randomized PTIME algorithm

```
  Given formula φ
y ← call FPRAs algorithm on φ with ε > 0
if y > 0 then
    output "satisfiable"
else
    output "unsatisfiable"
```

Correctness This works because:

- If φ is satisfiable, then $z \geq 1$ ($\#\varphi$) and $y > \frac{1}{1+\epsilon} > 0$. Therefore the algorithm would output "satisfiable."
- If φ is unsatisfiable, then $z = 0$ ($\#\varphi$) and $y = 0$. Therefore the algorithm would output "unsatisfiable."

In both cases, the algorithm outputs the correct response to a given Boolean formula φ with probability $\geq \frac{3}{4}$. Therefore, it seems unlikely that we can find an FPRAs for SAT. However, we can approximate #DNF in polynomial time.

4.2 Exact vs Approximate Counting

			Approx Counting
DNF	coNP-complete	#P-complete	polytime
CNF	NP-complete	#P-complete	hard
Perfect Matching	P	#P-complete	polytime
Spanning Trees	P	P	polytime

4.3 Approximate Counting for DNF

To accomplish this, we will use two main ideas:

- uniform generation of DNF sat assignments (algorithm 1)
- "downward self-reducibility" of DNF

Definition 6 (Downward Self-Reducible (DSR)) A problem is *downward self-reducible* if we can compute the problem by solving smaller and smaller subproblems and putting together answers via polytime computation.

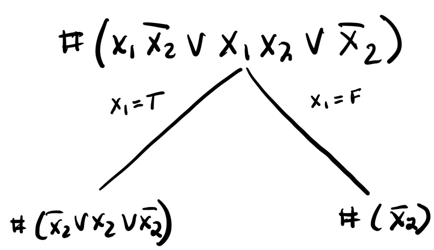
For #DNF, this means recursively solving smaller problems with one less variable. To put it more formally, #DNF is DSR because we can solve a given φ by computing:

$$\#\varphi(x_1, ..., x_n) = \#\varphi(x_1 = T, x_2, ..., x_n) + \#\varphi(x_1 = F, x_2, ..., x_n)$$

Both equations on the right-hand side are still DNFs but with $n - 1$ variables − basically by choosing the value of x_1, we can simplify φ.
Example

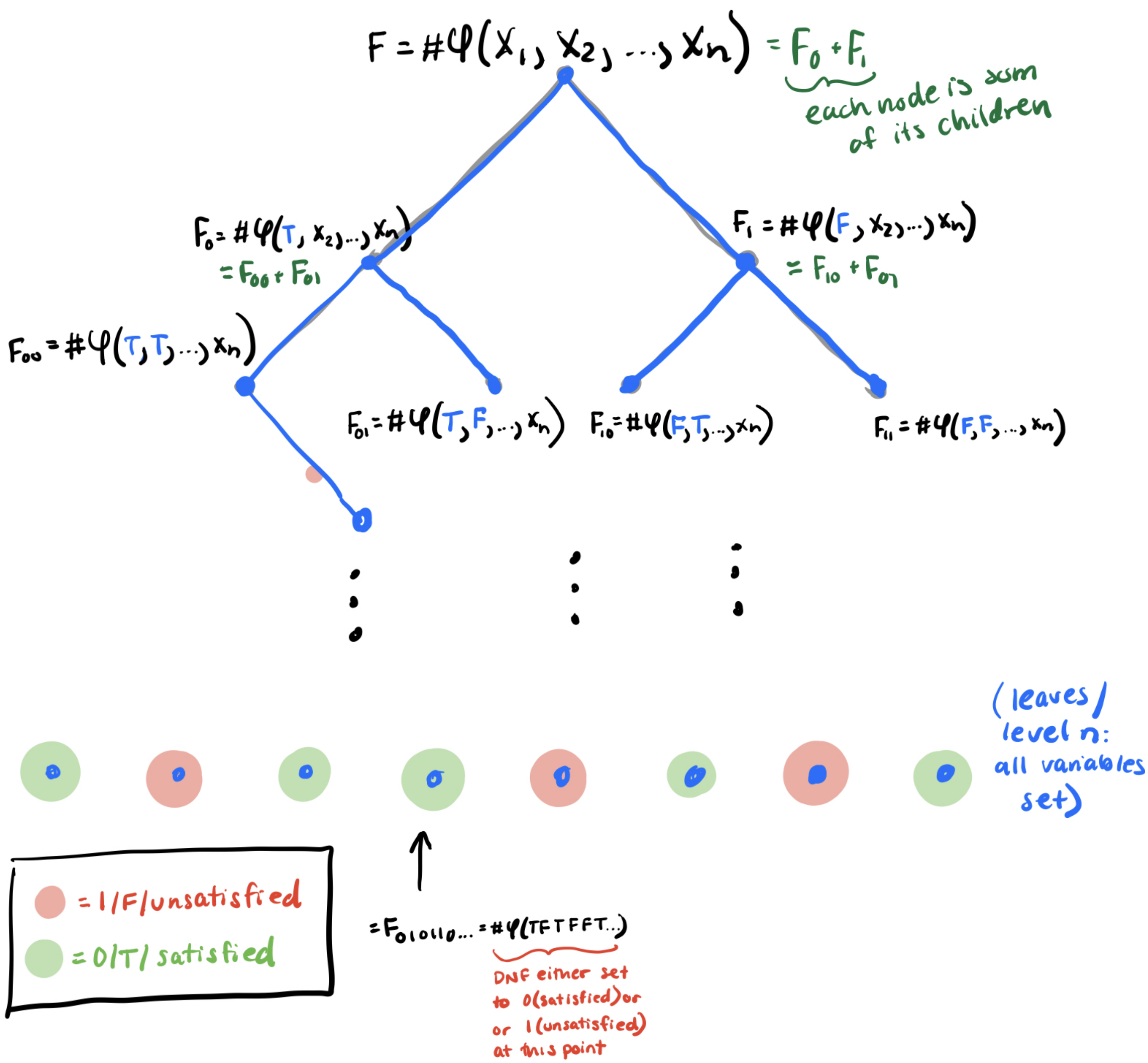
$$\#(x_1\bar{x_2} \vee x_1x_2 \vee \bar{x_2}) = \#(\bar{x_2} \vee x_2 \vee \bar{x_2}) + \#(\bar{x_2})$$

As we can see, the righthand side is constituted of the boolean function where $x_1 = T$ and the boolean function where $x_1 = F$. We can also represent this relation as a *downward self-reducibility tree*:



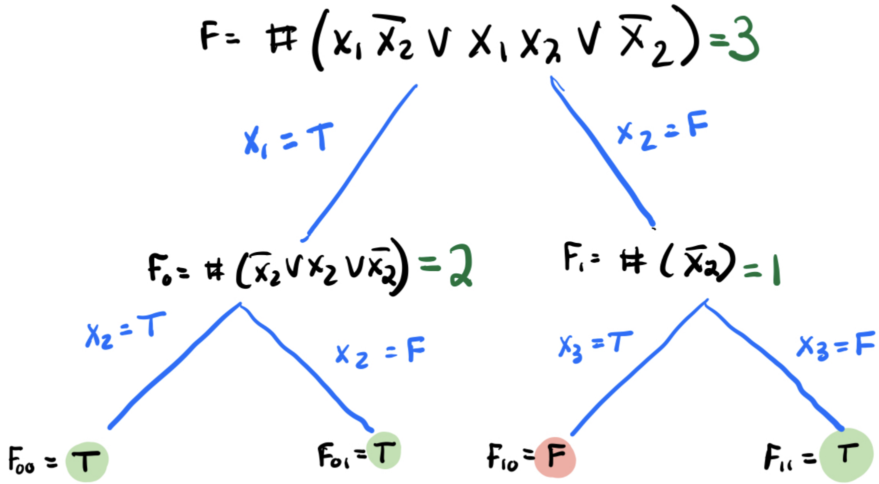
As we can see above, the node represent the equation given the settings of variables up until that point and the edges represent what the variable is set to. Each level i, where the root is level 0, represents possible configurations after setting x_i. We show a general form for this representation in the next section.

4.4 Downward Self-Reducibility Tree



As shown above, each node is the sum of its children, which have one more variable set. The leaves, therefore, are each possible way to set variables x_1 through x_n, and they either satisfy φ (colored green) and don't satisfy φ (colored red). Thereby, the number of satisfying assignments is just the number of green leaves.

Example



In this downward self-reducibility tree, we can see that the number of satisfying assignments to the earlier example is three.

4.5 Approximate Counting Algorithm for #DNF

Let $S_1 = \frac{F_1}{F} \to F = \frac{F_1}{S_1}$. S_1 is the fraction of satisfying assignments such that $x_1 = T$. Instead of trying to estimate F directly, we can estimate S_1 through sampling, and use that to find F. That's the basic idea behind this algorithm.