CS 49/149: 21st Century Algorithms (Fall 2018): Lecture 11

Date: 18th October, 2018

Topic: DNFs and Perfect Hashing

Scribe: Benedikt Bitterli

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to deeparnab@dartmouth.edu.

---

1 Randomized DNF counting

Suppose that you are given a formula ϕ in Disjunctive Normal Form (DNF). These formulas take the form:

$$\phi = x_1 x_2 \overline{x_3} + \overline{x_1} x_3 \overline{x_4} + \overline{x_4} x_5 x_6 + \cdots$$

A DNF formula consists of n variables $x_i \in \{T, F\}$ that can take on either true or false. These variables are arranged into m clauses, where each clause consists of a subset of the variables x_i (or their negation $\overline{x}_i$) combined using logical `and` (multiplication in the formula above). All clauses are combined using logical `or` ($+$ in the formula above).

A particular setting of all variables to true or false is called an *assignment*. If ϕ evaluates to true for a given assignment, we say that the assignment *satisfies* ϕ. There are 2^n possible assignments to the variables. You are now tasked with counting how many of these assignments satisfy the formula. How would you do it?

Call the number of assignments for which ϕ is satisfied N^*. A naive algorithm would simply iterate over all possible assignments and take exponential (2^n) time. But maybe we can do better using a randomized algorithm!

Try 1: Let's first try the straightforward randomized algorithm for this problem:

- Randomly set each x_i to true or false
- Check if $\mathbf{x}$ satisfies ϕ
- If ϕ is satisfied, return 2^n; else return 0

The return value of this algorithm is a random variable (call it X). What is the expected value of X?

$$\mathbb{E}[X] = 2^n \cdot \mathbb{P}[x \text{ satisfies } \phi] + 0 \cdot \mathbb{P}[x \text{ does not satisfy } \phi] \tag{1}$$

Our randomized algorithm samples an assignment uniformly at random. Therefore, the probability of obtaining a satisfying assignment is simply the number of satisfying assignments divided by the total number of assignments:

$$\mathbb{E}[X] = 2^n \cdot \frac{N^*}{2^n} = N^* \tag{2}$$

The expected value of this estimator is exactly the number of satisfying assignments! This means our simple algorithm is unbiased.

However, what will its variance be? Consider

$$\begin{aligned}\text{Var}[X] &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 &(3)\\ &= \left(2^{2n} \cdot \frac{N^*}{2^n}\right) - (N^*)^2 &(4)\\ &= N^* \cdot (2^n - N^*) &(5)\end{aligned}$$

If N^* is large, then this estimator is not terrible, since we are likely to sample a satisfying assignment. However, if there are few satisfying assignments, we are unlikely to sample one of them, and the variance becomes huge!

MULTIPLICATIVE BOUNDS

In the last lecture, we formulated the ϵ, δ bound in the form

$$\mathbb{P}[|Z - \mathbb{E}[Z]| > \epsilon] \leq \delta$$

We will refer to this as an *additive* bound, because we measure the absolute distance between Z and its expected value.

A different formulation is the *multiplicative* bound

$$\mathbb{P}[|Z - \mathbb{E}[Z]| > \epsilon \cdot \mathbb{E}[Z]] \leq \delta$$

This measures the *relative* error of Z to its expected value, which is more useful in some cases. Following the same proof as in the previous lecture, we can see that if Z is the average of t calls to a randomized algorithm X, then we need

$$\frac{\text{Var}[X]}{(\mathbb{E}[X])^2} \frac{1}{\epsilon^2} \ln \frac{1}{\delta} \tag{6}$$

calls to X to achieve the multiplicative ϵ, δ bound.

For the algorithm above, we can see that we need

$$\frac{N^* \cdot (2^n - N^*)}{(N^*)^2} \frac{1}{\epsilon^2} \frac{1}{\delta} = \left(\frac{2^n}{N^*} - 1\right) \frac{1}{\epsilon^2} \frac{1}{\delta}$$

calls to the algorithm to achieve the desired bound. Even if N^* is big (say, 1.99^n), the runtime is still exponential in n.

Our first algorithm sampled the space of possible assignments uniformly at random, which does poorly when the set of satisfying assignments is small. This means that we may be able to do better if we do a more "targeted" sampling of the space.

A DNF formula ϕ is satisfied when any one of its clauses are satisfied. For a given clause, how many assignments are there that satisfy it? Any variable that does not appear in the clause does not influence the outcome, and can take on true or false; on the other hand, any variable that does

appear in the clause must take on a fixed value (0 if it is negated, 1 if it is not). Therefore, if there appear k variables a clause, we can pick only the other $n-k$ variables freely, and there are 2^{n-k} satisfying assignments to the clause.

Let S_i be the set of satisfying assignments for clause i (as shown above, we can easily compute its size, $|S_i|$). Let U be the set of assignments that satisy ϕ. It is

$$U = \cup_{i=1}^{m} S_i \quad \text{and} \quad N^* = |U|$$

We might be tempted to simply sum up all the $|S_i|$ (which we can compute easily) to get N^*, but this is incorrect: An assignment x that satisfies ϕ might satisfy multiple clauses (e.g. clause i and j), and thus it appears in both S_i and S_j. Therefore, summing the sizes of all the sets might count some assignments multiple times, and we get an incorrect result.

However, this gives us an idea for an improved randomized algorithm: What if we assume the $|S_i|$ mostly don't overlap and return $\sum_i |S_i|$, but compensate for the overcounting using a randomized algorithm?

Try 2: Let $S = |\sum_i |S_i|$, and $N(x)$ be the number of clauses that a particular assignment x satisfies. Then our randomized algorithm works as follows:

1. Pick a clause i at random from $\{1, \ldots, m\}$ with some probability p_i
2. Pick an assignment x uniformly at random from S_i.
3. Calculate $N(x)$
4. Return $Z = S/N(x)$

This algorithm returns the naive sum of the sizes of $|S_i|$, but compensates for overcounting by downweighting the sum when we find an assignment x that satisfies multiple clauses (and is therefore overcounted).

Is this estimator unbiased? Let's compute its expected value:

$$\begin{aligned}
\mathbb{E}[Z] &= \sum_{x \in U} \frac{S}{N(x)} \mathbb{P}[x \text{ is sampled}] \\
&= \sum_{x \in U} \frac{S}{N(x)} \sum_{S_i : x \in S_i} \mathbb{P}[S_i \text{ is sampled in step 1}] \cdot \mathbb{P}[x \text{ is sampled in step 2}] \\
&= \sum_{x \in U} \frac{S}{N(x)} \sum_{S_i : x \in S_i} p_i \cdot \frac{1}{|S_i|}
\end{aligned}$$

So far, we have not specified what p_i should be. However, looking at the above equation, we can see that we are summing over all elements in U. Remember that $N^* = |U|$; therefore, we want to cancel all inner terms so that the sum simply counts the size of U. Therefore, we set $p_i = |S_i|/S$, i.e. we pick sets with probability proportional to their size.

This gives us

$$\begin{aligned}\mathbb{E}[Z] &= \sum_{x \in U} \frac{S}{N(x)} \sum_{S_i : x \in S_i} \frac{|S_i|}{S} \cdot \frac{1}{|S_i|} \\ &= \sum_{x \in U} \frac{1}{N(x)} \sum_{S_i : x \in S_i} 1 \\ &= \sum_{x \in U} \frac{1}{N(x)} N(x) \\ &= \sum_{x \in U} 1 \\ &= |U| = N^*\end{aligned}$$

This makes our algorithm an unbiased estimator!

To compute the variance of this estimator, we make use of the following fact: ✍

Fact 1: *For a random variable Z that never exceeds $\max Z$, it holds that $\mathrm{Var}[Z] \leq (\max Z)\mathbb{E}[Z]$*

We know that $N(x) \geq 1$, and thus $Z \leq S = \sum_i |S_i| \leq m \cdot |\cup_i S_i| = m \cdot \mathbb{E}[Z]$. Then

$$\mathrm{Var}[Z] \leq (\max Z)\mathbb{E}[Z] \leq m \cdot (\mathbb{E}[Z])^2$$

If we want to achieve a multiplicative ϵ, δ bound with this algorithm, then we need to average

$$\frac{\mathrm{Var}[Z]}{(\mathbb{E}[Z])^2} \frac{1}{\epsilon^2} \frac{1}{\delta} \leq m \cdot \frac{1}{\epsilon^2} \frac{1}{\delta}$$

calls to this algorithm. This bound only depends on m, the number of clauses, and is a tremendous improvement over our first algorithm.

SAMPLING A POINT IN S_i

The algorithm we just saw relies on the ability to sample a valid assignment to clause i uniformly at random. But we can easily do this: Variables that don't appear in the clause don't influence its result, and we can randomly set them to true or false. Variables that do appear in the clause can only take on one possible value if the clause is to be satisfied, and we simply set them to that value (0 if they appear negated, 1 otherwise).

2 Perfect Hashing

Suppose we are given a universe U of words, and a *static* dictionary $D \subseteq U$ (i.e. it is fixed at the beginning and does not change) of size $s = |D|$. The size of the universe is much larger than that of the dictionary, i.e. $|U| \gg s$.

Further, suppose we need to answer queries of the form "Is $x \in D$?" for any $x \in U$. How can we answer such queries efficiently?

A *hash table* is a data structure that solves this problem. A hash table maintains an array of size n, where each entry (also called "bucket") stores a linked list of items. Associated with the hash table is a *hash function* $h : U \to \{1, \ldots, n\}$ that maps points in the universe to buckets in the array. When we build the hash table, we compute $h(x)$ for each $x \in D$ and append x to the list at the array index $h(x)$. We can also answer queries "Is $x \in D$?" by computing $h(x)$ and checking if x appears in the list at that index.

Searching a bucket takes linear time, so the runtime of queries depends on how many items end up in the same bucket. To make hash tables efficient, we need to find a hash function that prevents too many items being hashed to the same value.

UNIVERSAL HASH FAMILY

A universal hash family (UHF) H is a set of hash functions h s.t. for any $x, y \in U$, $x \neq y$:

$$\mathbb{P}_{h \in H}[h(x) = h(y)] \leq \frac{1}{n}$$

PAIRWISE INDEPENDENT HASH FAMILY

A pairwise independent hash family (PIHF) H is a set of hash functions h s.t. for any $x, y \in U$, $x \neq y$, and for any $i, j \in \{1, \ldots, n\}$:

$$\mathbb{P}_{h \in H}[h(x) = i \wedge h(y) = j] \leq \frac{1}{n^2}$$

✍

Exercise: *Prove that a PIHF is also a UHF*

2.1 Perfect Hashing

We will now look at an algorithm that finds a hash function for the static dictionary problem that achieves zero collisions. This is called "perfect" hashing.

Suppose you are given a universal hash family H. We begin by picking a hash function h uniformly at random from H, and build the corresponding hash table. Note that the hash function is picked once and then does not change as new queries come in - that is, all the randomness happens at the beginning. The queries themselves are deterministic once the hash function is fixed.

Suppose a query comes along that searches for item q. The time taken to process q (call it Time(q)) is equivalent to the number of items in D that hash to the same value as $h(q)$. Because we picked h at random, Time(q) is a random variable. What is its expected value?

$$\begin{aligned}\mathbb{E}_{h \in H}[\text{Time}(q)] &= \mathbb{E}_{h \in H}[\#\text{of items in } D \text{ that collide with } q] \\ &= \mathbb{E}_{h \in H}[|\{x \in D : h(x) = h(q)\}|]\end{aligned}$$

Now define an indicator variable for collisions between x and q:

$$A_{xq} = \begin{cases} 1 & \text{if } h(x) = h(q) \\ 0 & otherwise \end{cases}$$

Then

$$\begin{aligned}\mathbb{E}_{h\in H}[\text{Time}(q)] &= \mathbb{E}_{h\in H}\left[\sum_{x\in D} A_{xq}\right] \\ &= \sum_{x\in D} \mathbb{E}_{h\in H}[A_{xq}] \\ &= \sum_{x\in D} \mathbb{P}_{h\in H}[h(x) = h(q)] = \frac{|D|}{n}\end{aligned}$$

We used that fact that h was picked from a UHF to obtain the last step. This result means that if we set n to be on the order of $|D|$, we get $\mathbb{E}_h[\text{Time}(q)] = O(1)$ for any query q!

Is expected constant time enough? It may be that this hash family does well on average, but there are no guarantees on the *worst case* time. It may be that we get unlucky and get a hash function that does poorly.

Let's try to bound the worst case time. The longest query time is certainly bounded by the total number of collisions in the hash table, which is $\sum_{x,y\in D} A_{xy}$. What can we say about this term?

$$\begin{aligned}\mathbb{E}_{h\in H}[\#\text{collisions}] &= \mathbb{E}_{h\in H}\left[\sum_{x,y\in D, x\neq y} A_{xy}\right] \\ &= \sum_{x,y\in D, x\neq y} \mathbb{E}_{h\in H}[A_{xy}] \\ &= \frac{1}{n}\cdot \#\text{of pairs in D} \\ &= \frac{1}{n}\binom{s}{2} \leq \frac{s^2}{2n}\end{aligned}$$

If we choose $n = s^2$, then we obtain $\mathbb{E}_{h\in H}[\#\text{collisions}] \leq 1/2$.

What is the probability of a hash function generating more than one collision? We can apply the Markov bound to obtain

$$\mathbb{P}_{h\in H}[\#\text{collisions} > 1] \leq \frac{\mathbb{E}_{h\in H}[\#\text{collisions}]}{1} \leq \frac{1}{2}$$

This means the probability of having *any* collision at all is less than $1/2$.

This is great! This allows for a trivial algorithm to obtain a hash table with no collisions: Pick an h from H at random, and generate the hash table. If there is a collision, throw away h and pick a new one. Because the probability of a "bad" event is at most $1/2$, we need to repeat this procedure only twice on average before obtaining a perfect hash table.

Remark: *This hash table is "perfect" in terms of runtime, but very inefficient in space: We need to pick its size as s^2, which is huge. We can do better by hashing twice: First, generate a hash table of size s with a "low" number of collisions; then hash each bucket into its own (collision-free) hash table with size set to be the squared sized of the bucket.*