Lecture 7: TQBF

Anup Rao

February 16, 2022

The TQBF function maps the set of totally quantified boolean formulas to 0 or 1. A totally quantified boolean formula is something that looks like this:

$$\psi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_n \phi(x_1, \ldots, x_n),$$

where here ϕ is a boolean formula on the variables $x_1, \ldots, x_n$.

$$\mathsf{TQBF}(\psi) = 1 \text{ if and only } \psi \text{ is true.}$$

TQBF's help characterize **PSPACE**.

Lemma 1. *$\mathsf{TQBF}(\psi)$ for $\psi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_n \phi(x_1, \ldots, x_n)$ can be computed in space $O(m \cdot n)$, where size of ϕ is m. In other words, $\mathsf{TQBF} \in$ **PSPACE**.*

Proof First note that for every fixing of $x_1, \ldots, x_n$, ϕ can be computed in space $O(m)$. Let

$$A = \forall x_2 \exists x_3 \cdots \exists x_n \phi(0, x_2, \ldots, x_n)$$

and

$$B = \forall x_2 \exists x_3 \cdots \exists x_n \phi(1, x_2, \ldots, x_n).$$

We know that $\mathsf{TQBF}(\psi) = \mathsf{TQBF}(A) \vee \mathsf{TQBF}(B)$ (similarly we will have to compute $A \wedge B$ when the first quantifier is a $\forall$). Writing down A takes at most $O(m)$ space. Let $S(n)$ denote the space required to compute $\mathsf{TQBF}(\psi)$. Now, computing A recursively uses $S(n-1)$ space. After computing A, we can store the answer (one bit) and erase all contents of the tape that was used to compute A. We then write down B and compute $\mathsf{TQBF}(B)$ recursively. Overall, we have that $S(n) = S(n-1) + O(m)$. As we know that $S(0) = O(m)$, we can conclude that $S(n) = O(m \cdot n)$. ■

Theorem 2. *For every boolean $f \in$ **PSPACE**, there is a polynomial time computable function g mapping bits to truly quantified boolean formulas such that $f(x) = \mathsf{TQBF}(g(x))$.*

Proof We shall show how to use the formula to encode connectivity in the configuration graph of the machine that computes f. This is a graph of size $2^t = 2^{\text{poly}(n)}$.

We generate a formula $\psi_i(A, B)$ in $\text{poly}(n)$ time that checks whether there is a path of length $\leq 2^i$ from A to B. When $i = 0$. $\psi_i(A, B)$ just needs to check that B is the configuration that comes after A. Since we know that there is a polynomial sized circuit $\mathcal{C}$ such that $\mathcal{C}(x, A)$ computes the configuration that follows from A, we can construct a circuit $\mathcal{F}$ of size poly(n) such that

$$\mathcal{F}(A, B, x) = \begin{cases} 1 & \text{if } \mathcal{C}(A, x) = B, \\ 0 & \text{else.} \end{cases}$$

Just like in the proof that SAT is **NP**-complete, we can generate a polynomial sized formula $F(y)$ such that $\exists y F(y)$ is true if and only if $\mathcal{F}(A, B, x) = 1$.

For the general case, note that there is a path of length at most 2^i from A to B if and only if there is some vertex C in the graph such that there is a path of length 2^{i-1} from A to C and a path of length 2^{i-1} from C to A. Thus we can define

$$\psi_i(A, B) = \exists C, \psi_{i-1}(A, C) \wedge \psi_{i-1}(C, B).$$

However, this doubles the size of the formula ψ_{i-1} (which means that after t steps we will be trying to generate a formula that is exponentially big and this is impossible in polynomial time).

Indeed, we haven't yet used the $\forall$ quantifiers. Let us use the same idea as before to define the smaller formula:

$$\begin{aligned} &\psi_i(A, B) \\ &= \exists C, \forall X, \forall Y, (X = A \wedge Y = C) \vee (X = C \wedge Y = B) \Rightarrow \psi_{i-1}(X, Y) \\ &= \exists C, \forall X, \forall Y, (\neg(X = A \wedge Y = C) \wedge \neg(X = C \wedge Y = B)) \vee \psi_{i-1}(X, Y) \end{aligned}$$

The end result is a formula of size poly(n, t) that checks for a path of length 2^t in the graph as required. ■

Lower Bounds on SAT

The material in this section was not discussed in class. We include it here as you might find it interesting. Although we cannot say anything non-trivial about the running time required to compute SAT, or the space required to compute SAT, we can show that SAT cannot have an algorithm that is both linear time and log space:

Theorem 3. *There is no turing machine computing* SAT *in* $O(n)$ *time and* $O(\log n)$ *space.*

In order to prove the theorem, we shall rely on two facts that we have convinced ourselves of before:

Theorem 4. *If $t(n) \geq \Omega(n)$, any $f \in \mathsf{NTIME}(t(n))$ can be reduced in in logarithmic space and time $O(t(n)\log(t(n)))$ to computing* SAT *on a formula of size $O(t(n)\log t(n))$.*

Earlier in the course we proved that the reduction is in polynomial time, but in fact it is even in **L**. (Think about this!). The reduction works by first computing a circuit that simulates the computation of a machine, and then computing the formula that simulates the execution of the circuit.

Another theorem we shall appeal to is the deterministic time hierarchy theorem:

Theorem 5 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n)\log r(n) = o(t(n))$, then $\mathsf{DTIME}(r(n)) \subsetneq \mathsf{DTIME}(t(n))$.*

Proof of Theorem 3: Assume for the purpose of contradiction that there is a turing machine computing SAT in $O(n)$ time and $O(\log n)$ space. The idea is to use the purported SAT algorithm to get an unreasonable speed up of computations. Suppose for the sake of contradiction that SAT can be computed in linear time and logarithmic space.

Suppose that $f \in \mathsf{DTIME}(n^2)$ via the machine M_f and $f \notin \mathsf{DTIME}(n\mathsf{polylog}(n))$. Such an f exists by Theorem 5. We shall show how to compute f in time $O(n\mathsf{polylog}(n))$, giving us the desired contradiction.

By appealing to Theorem 4, consider the machine M that runs as follows on input $x \in \{0,1\}^n$:

1. Generate the formula ϕ of size $n^2 \log n$ that simulates the machine $M_f(x)$, using Theorem 4.
2. Check whether $M_f(x)$ accepts by computing $\mathsf{SAT}(\phi)$ in time $O(n^2 \log n)$ and space $O(\log(n^2 \log n)) = O(\log n)$.

M is not our final simulation. M computes f in time $O(n^2 \log n)$ and space $O(\log n)$.

Consider the configuration graph of M. This graph accepts if and only if there is an accepting path of length $t = O(n^2 \log n)$, which happens if and only if there exist $\sqrt{t}$ intermediate configurations $C_1, \ldots, C_{\sqrt{t}}$, such that there is a path of length $\sqrt{t}$ between intermediate configurations. In other words, $f(x) = 1$ if and only if

$$\exists C_1, \ldots, C_{\sqrt{t}}, \forall i, C_i \text{ follows from } C_{i-1} \text{ in } \sqrt{t} \text{ steps.}$$

Each configuration takes only $O(\log n)$ bits to write down. So once we guess all of these $\sqrt{t}$ configurations, the problem of determining whether they determine an accepting of path of length t can be encoded using a SAT formula of size $O(\sqrt{t} \cdot \log n \cdot \mathsf{polylog}(t,n))$ (by Theorem 4), so it can be solved in deterministic time $O(\sqrt{t} \cdot \mathsf{polylog}(t,n))$.

Thus we can compute a formula ψ of size $O(\sqrt{t} \cdot \mathsf{polylog}(t,n))$ such that $f(x) = 1$ if and only if

$$\exists C_1, \ldots, C_{\sqrt{t}}, \exists z, \psi(C_1, \ldots, C_{\sqrt{t}}, z).$$

The above is an instance of SAT and can then be solved deterministically in time $O(\sqrt{t} \cdot \mathsf{polylog}(n,t))$. Thus, overall, we get a simulation in deterministic time $O(\sqrt{t} \cdot \mathsf{polylog}(t,n)) = O(n\mathsf{polylog}(n)) = o(n^2)$, contradicting the deterministic time hierarchy theorem. ■

Randomized Algorithm review

WE DID NOT DISCUSS this material in class. I include it here for your reference:

Probability Spaces

A *probability space* is a set Ω such that every element $a \in \Omega$ is assigned a number $0 \leq \Pr[a] \leq 1$ (called the probability of a), and $\sum_{a \in \Omega} \Pr[a] = 1$.

An *event* in this space is a subset $E \subseteq \Omega$. The probability of the event is $\sum_{a \in E} \Pr[a]$. For example, imagine we toss a fair coin n times. Then the probability space consists of the 2^n possible outcomes of the coin tosses. If E is the event that the first k coin tosses are heads, this event has probability exactly 2^{-k}. Given two events E, E', we write $\Pr[E|E']$ to denote $\Pr[E \cap E'] / \Pr[E']$. This is the probability that E happens given that E' happens. We say that E, E' are independent if $\Pr[E \cap E'] = \Pr[E] \cdot \Pr[E']$. In other words, E, E' are independent if $\Pr[E|E'] = \Pr[E]$.

A *real valued random variable* is a function $X : \Omega \to \mathbb{R}$. The number of heads in the coin tosses is a random variable. The expected value of a random variable X is defined as $\mathbb{E}[X] = \sum_{a \in \Omega} \Pr[a] \cdot X(a)$. The following lemma is a very useful fact about random variables.

Lemma 6 (Linearity of expectation)**.** *If X, Y are real random variables, then $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.*

Proof

$$\begin{aligned}
\mathbb{E}[X+Y] &= \sum_{a \in \Omega} \Pr[a] \cdot (X(a) + Y(a)) \\
&= \sum_{a \in \Omega} \Pr[a] \cdot Y(a) + \sum_{a \in \Omega} \Pr[a] \cdot X(a) \\
&= \mathbb{E}[X] + \mathbb{E}[Y].
\end{aligned}$$

■

For example, let us calculate the expected number of runs of seeing 7 contiguous heads or tails in a 200 coin tosses. Let X_i be 1 if there are 7 heads or tails that start at the i'th position, and 0 otherwise. If $1 \leq i \leq 194$, then $\mathbb{E}[X_i] = \Pr[X_i = 1] = 2 \cdot 2^{-7} = 1/64$. If $i \geq 196$, then $X_i = 0$. On the other hand, the total number of such runs is $\sum_{i=1}^{194} X_i$. So by linearity of expectation, the expected number of such runs is $194/64 \approx 3.031$.

Here is an expectation basic magic trick: Tell your audience to generate two sequences of coin tosses—one generated using 200 flips of a coin, and the second generated by hand. You leave the room, and they write both sequences on a black board. Then you come back into the room and immediately point out the sequence that was generated by hand. The trick: a random sequence is very likely to have a run of 7 heads or tails, while people tend to not insert such a long run into a sequence that they think looks random.

In class, we discussed the waiting time to see the first heads. Suppose you keep tossing a fair coin until you see heads. Let T be the number of tosses you make. What is the expected value of T? The key observation is that if the first toss is a heads, you stop with $T = 1$. Otherwise, the rest of the experiment is exactly the same as the original random experiment. So, we get:

$$
\begin{aligned}
&\mathbb{E}[T] = (1/2) \cdot 1 + (1/2) \cdot (1 + \mathbb{E}[T]) \\
\Rightarrow &\mathbb{E}[T] \cdot (1 - 1/2) = 1 \\
\Rightarrow &\mathbb{E}[T] = 2.
\end{aligned}
$$

Randomized Algorithms

We shall give a few examples of problems where randomness helps to give very effective solutions.

Matrix Product Checking

Suppose we are given three $n \times n$ matrices A, B, C, and want to check whether $A \cdot B = C$. One way to do this is to just multiply the matrices, which will take much more than n^2 time. Here we give a randomized algorithm that takes only $O(n^2)$ time.

Input: 3 $n \times n$-matrices A, B, C
Result: Whether or not $A \cdot B = C$.
Sample an n coordiante column vector $r \in \{0,1\}^{0,1}$ uniformly at random ;
if *A(B(r)) = C(r)* **then**
  Output "Equal";
else
  Output "Not equal";
end

Algorithm 1: Algorithm for Multiplication Checking

The algorithm only takes $O(n^2)$ time. For the analysis, observe that if $AB = C$, then the algorithm outputs "Equal" with probability 1. If $AB \neq C$, the algorithm outputs "Equal" only when $ABr = Cr \Rightarrow (AB - C)r = 0$. We shall show that this happens with probability at most 1/2.

Let $D = AB - C$. Then $D \neq 0$, so let d_{ij} be a non-zero entry of D. Then we have that the i'th coordinate $(Dr)_i = \sum_k d_{ik} \cdot r_k$. This coordinate is 0 exactly when $r_j = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k$. Finally, observe

$$
\begin{aligned}
&\Pr\left[r_j = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&= \sum_a \Pr\left[a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \cdot \Pr\left[r_j = a | a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&\leq 1/2 \sum_a \Pr\left[a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&= 1/2.
\end{aligned}
$$

Exercise: Modify the above algorithm so that the probability the algorithm outputs "Equal" when $AB \neq C$ is at most 1/4.

2-SAT

A two SAT formula is a CNF formula where each clause has exactly 2-variables. Here we give a randomized algorithm that can find a satisfying assignment to such a formula, if one exists.

```
Input: A two sat formula φ
Result: A satisfying assignment for φ if one exists
Set a = 0 to be the n-bit all 0 string;
for i = 1, 2, ..., 100n^2 do
    if φ(a) = 1 then
        Output a;
    end
    Let a_i, a_j be the variables of an arbitrary unsatisfied clause.
     Pick one of them at random and flip its value ;
end
Output "Formula is not satisfiable";
```

Algorithm 2: Algorithm for 2 SAT

If ϕ is not satisfiable, then clearly the algorithm has a correct output. Now suppose ϕ is satisfiable and b is a satisfying assignment, so $\phi(b) = 1$. We claim that the algorithm will find b (or some other satisfying assignment) within $100n^2$ steps with high probability. To understand the algorithm, let us keep track of the number of coordinates that a, b disagree in during the run of the algorithm. Observe that during each run of the for loop, the algorithm picks a clause that is unsatisfied under a. Since b satisfies this clause, a, b must disagree in one of the two variables of this clause. Thus the algorithm reduces the distance from a to b with probability $1/2$.

Thus we can think of the algorithm as doing a random walk on the line. There are $n+1$ points on the line, and at each step, if the algorithm is at position i it moves to position $i+1$ with probability $1/2$ and to position $i-1$ with probability at least $1/2$. We are interested in the expected time before the algorithm hits position 0. Let

$$t_i = \mathbb{E}\left[\text{\# steps before hitting position 0 from position } i\right].$$

Then we have the following equations:

$$\begin{aligned} t_0 &= 0, \\ t_i &= (1/2)t_{i+1} + (1/2)t_{i-1} + 1 \qquad\qquad i \neq 0, n \\ \Rightarrow t_i - t_{i-1} &= t_{i+1} - t_i + 2 \\ t_n &= 1 + t_{n-1}. \end{aligned}$$

Thus we can compute:

$$\begin{aligned} t_n &= (t_n - t_{n-1}) + (t_{n-1} - t_{n-2}) + \ldots + (t_1 - t_0) \\ &= 1 + 3 + \ldots \\ &= \sum_{j=1}^{n} (2j-1) = 2\left(\sum_{j=1}^{n} j\right) - n = n(n+1) - n = n^2. \end{aligned}$$

Thus the expected time for the algorithm to find a satisfying assignment is n^2.

Lemma 7.

$$\Pr[\textit{algorithm does not find satisfying assignment in } 100n^2 \textit{ steps}] < 1/100.$$

Proof We have that

$$\begin{aligned} n^2 &\geq \mathbb{E}\left[\text{\# steps to find assignment}\right] \\ &= \sum_{s=0}^{\infty} s \cdot \Pr[s \text{ steps to find assignment}] \\ &\geq \Pr[\text{at least } 100n^2 \text{ steps are taken}] \cdot 100n^2. \end{aligned}$$

Therefore,

$$\Pr[\text{more than } 100n^2 \text{ steps are taken}] < 1/100.$$

■

Max Cut

Given a graph $G = (V, E)$, a subset $S \subset V$ is called a cut of the graph. The size of the cut is the number of edges that cross from S to $V - S$. It is known to be NP-hard to compute the MAX-cut of a graph. Here we give a simple randomized algorithm that will compute a cut that is half as big as the biggest cut in expectation.

The algorithm is just to pick the subset S at random, by including every vertex in S with probability half. For each edge e, let X_e be the random variable that is 1 if e goes from S to $V - S$, and 0 otherwise. Then we see that the size of the cut is exactly $\sum_{e \in E} X_e$. We can compute $\mathbb{E}[X_e] = 1/2$, and so by linearity of expectation,

$$\mathbb{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbb{E}[X_e] = |E|/2.$$

Fingerprinting

Suppose Alice has an n-bit string x and Bob has an n-bit string y, and they want to check that they are equal. Naively this takes n bits of communication between them. We can do much better using randomization.

Alice samples a random prime number p from the set of primes that are less than $cn \ln n$, for some constant c that we shall pick later. She then sends p and $x \mod p$ to Bob. Bob checks that $x \mod p$ is equal to $y \mod p$. Thus they only need to communicate $O(\log n)$ bits in this process.

If $x = y$, this will always produce the right outcome. We shall argue that if $x \neq y$, the probability that they make a mistake is going to be very small. To do this, we need a theorem:

Theorem 8 (Prime number theorem)**.** *Let $\pi(a)$ denote the number of primes that are at most a. Then $\lim_{a \to \infty} \frac{\pi(a)}{a/\ln a} = 1$.*

When $x \neq y$, the above process fails only when p divides $x - y$. Since $|x - y| \leq 2^n$, $x - y$ can have at most n prime factors. On the other hand, by the prime number theorem, the number of primes of size up to $cn \ln n$ is at least $cn \ln n/(\ln(cn \ln n)) = \Omega(cn)$. Thus the probability that the prime Alice picks divides $x - y$ is at most $O(1/c)$.

Randomized Classes

There are several different ways to define complexity classes involving randomness. A turing machine with access to randomness is just like a normal turing machine, except it is allowed to toss a random coin in each step, and read the value of the coin that was tossed.

BPP

We say that the randomized machine computes the function f if for every input x, $\Pr_r[M(x,r) = f(x)] \geq 2/3$, where the probability is taken over the random coin tosses of the machine M. **BPP** is the set of functions that are computable by polynomial time randomized turing machines in the above sense.

RP

We shall say that $f \in$ **RP** if there is a randomized machine that always compute the correct value when $f(x) = 0$, and computes the correct value with probability at least $2/3$ when $f(x) = 1$.

ZPP

Finally, we define the class **ZPP** to be the set of boolean functions that have an algorithm that *never* makes an error, but whose *expected* running time is polynomial in n.