COSC 544 Probabilistic Proof Systems 10/5/17

Turning Computer Programs Into Circuits (Part 2)

Lecturer: Justin Thaler

1 Overview of This Lecture

In the previous lecture, we say two methods for turning computer programs into arithmetic circuits. The first method was undesirable for two reasons. First, it yielded circuits of very large depth (so large, in fact, that applying the GKR protocol to the resulting circuits led to a verifier runtime that was as bad as just having the verifier run the entire program without any help from a prover). Second, if the computer program ran in time T and space s, the circuit had size at least $T \cdot s$, and we'd really prefer to have circuits with close to T gates.

In this lecture, we are going to address both of these issues. However, to do so, we are going to have to shift from talking about circuit *evaluation* to talking about circuit *satisfiability*.

In the arithmetic circuit evaluation problem considered in the previous lectures, the input specifies an arithmetic circuit $\mathcal{C}$, input x, and output(s) y, and the goal is to determine whether $\mathcal{C}(x) = y$. In the arithmetic circuit *satisfiability* problem, the circuit $\mathcal{C}$ takes *two inputs* x and w. The second witness w is often called the *witness*, or sometimes the *non-deterministic input*. Given the first input x and output(s) y, the goal is to determine whether *there exists* a w such that $\mathcal{C}(x,w) = y$.

The purpose of this lecture is to explain that any computer program running in time T can be efficiently transformed into an instance $(\mathcal{C},x,y)$ of arithmetic circuit satisfiability, where the circuit $\mathcal{C}$ has size close to T, and depth close to $\log T$. That is, the output of the program on input x equals y if and only if there exists a w such that $\mathcal{C}(x,w) = y$. Moreover, any party (such as a prover) who actually runs the program on input x can easily construct a w satisfying $\mathcal{C}(x,w) = y$.

Preview: Why Turning Computer Programs Into Instances of Circuit Satisfiability Is Useful. Why is such a transformation of a high-level computer program to an equivalent instance $(\mathcal{C},x,w)$ of circuit satisfiability useful for designing efficient interactive proofs?

One way we could design an efficient interactive proof for determining the output of the program on input x is to have the prover send a witness w to the verifier, and run the GKR protocol to efficiently check that $\mathcal{C}(x,w) = y$. This would be enough to convince the verifier that indeed the program outputs y on input x. This approach works well if the witness w is small. But in the computer-program-to-circuit-satisfiability transformation that we're about to see, the witness w will be very large (of size roughly T, the runtime of the computer program). So even asking the verifier to read the claimed witness w is as expensive as asking the verifier to simply run the program herself without the help of a prover.

Fortunately, we will see next lecture that we can combine the GKR protocol with cryptography to avoid having the prover send the entire witness w to the verifier.

Specifically, recall that in order to run the GKR protocol on circuit $\mathcal{C}$ with input $u := (x,w)$, the only information about the input that the verifier needs to know is the evaluation of the multilinear extension $\tilde{u}$ of u at a random point, and that this evaluation is only needed by the verifier at the very end of the protocol.

We will explain next lecture that in order to quickly evaluate $\tilde{u}$ at any point, it is enough for the verifier to know the evaluation of the multilinear extension $\tilde{w}$ of w at a related point.

Rather than having the prover explicitly send the witness w to the verifier, and then applying the GKR protocol to ensure that indeed $\mathcal{C}(x,w) = y$, we will instead have the prover send a short *cryptographic commitment* to the multilinear polynomial $\tilde{w}$. This commitment effectively *binds* the prover to a specific multilinear polynomial $\tilde{w}$, so that the verifier can later ask the prover to tell her $\tilde{w}(r)$ for some point r (i.e., asks the prover to *decommit* to $\tilde{w}$ at input r), the prover will not be able to "change" its answer in a manner that depends on r.

After the prover commits to the polynomial $\tilde{w}$, we will run the GKR protocol to check that $\mathcal{C}(x,w) = y$. The verifier can happily run this protocol even though it doesn't know w, until the very end of the protocol when it has to evaluate $\tilde{u}$ at a single point r (and hence at this point the verifier needs to know $\tilde{w}(r)$). The verifier learns $\tilde{w}(r)$ from the prover, by having the prover decommit to $\tilde{w}$ at input r.

All told, this approach will lead to an *argument* system that effectively forces a prover to run an arbitrary computer program on an input x. The argument system is nearly optimal in the sense that the verifier runs in linear time in the size of the input, and the prover runs in time close to T, where T is the time required to simply run the program on input x. $\square$

2 The Transformation From Computer Programs To Arithmetic Circuit Satisfiability

Before describing the transformation, it is helpful to consider why the circuit generated in Method 1 of the previous lecture had at least $T \cdot s$ gates, which is significantly larger than T if s is large. The answer is that that circuit consisted of T "stages" where the ith stage computed each bit of the machine's configuration (which includes the entire contents of its main memory) after i machine instructions had been executed.

But each machine instruction affects the value of only $O(1)$ registers and memory cells, so between any two stages, almost all bits of the configuration remained unchanged. This means that almost all of the gates and wires in the circuit are simply devoted to copying bits from the configuration after i steps to the configuration after step $i+1$. Intuitively, this is highly wasteful, and in order to obtain a circuit of size close to T, rather than $T \cdot s$, we will need to cut out all of this redundancy.

To describe the main idea in the transformation, it is helpful to introduce the notion of the *transcript* (sometimes also called a *trace*) of a random access machine M's execution on input x. Roughly speaking, the transcript describes just the *changes* to M's configuration at each step of its execution. That is, for each step i that M takes, the transcript lists just the value of each register and the program counter at the end of step i. Since M has only $O(1)$ registers, the transcript can be specified using $O(T)$ words (where a word refers to a value that can be stored in a single register or memory cell).

The basic idea is that the transformation from RAM execution to circuit satisfiability produces a circuit satisfiability instance $(\mathcal{C},x,y)$, where x is the input to M, y is the claimed output of M, and the witness w is supposed to be the transcript of M's execution of input x. The circuit $\mathcal{C}$ will simply check that w is indeed the transcript of M's execution on input x, and if this check passes, then $\mathcal{C}$ outputs the same value as M does according to the ending configuration in the transcript If the check fails, $\mathcal{C}$ outputs a special rejection symbol.

A schematic of $\mathcal{C}$ is depicted in Figure 1.

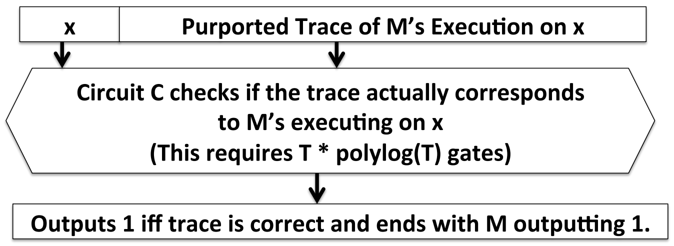
x
Purported Trace of M's Execution on x
Circuit C checks if the trace actually corresponds to M's executing on x (This requires T * polylog(T) gates)
Outputs 1 iff trace is correct and ends with M outputting 1.

Figure 1: Sketch of the transformation From RAM execution on input x to an instance of circuit satisfiability.

2.1 Details of the Transformation

The circuit $\mathcal{C}$ takes an entire transcript (sorted by time) of the entire execution of M as a non-deterministic input, where a transcript consists of (timestamp, list) pairs, one for each step taken by M. Here, a list specifies the bits contained in the current program counter and the values of all of M's registers. The circuit then checks that the transcript is valid. This requires checking the transcript for both time consistency (i.e., that the claimed state of the machine at time i correctly follows from the machine's claimed state at time $i-1$) and memory consistency (i.e., that whenever a value is read from memory location, the value that is returned is equal to the last value written to that location).

The circuit checks time-consistency by representing the transition function of the RAM as a small sub-circuit. It then applies this sub-circuit to each entry i of the transcript and checks that the output is equal to entry $i+1$ of the transcript.

The circuit checks memory consistency by re-sorting the transcript based on memory location (with ties broken by time), at which point it is straightforward for the circuit to check that every memory read from a given location returns the last value written to that location. The sorting step is the most conceptually involved part of the construction of $\mathcal{C}$, and works as follows.

Note that all of the T time-consistency checks and (at most) T memory-consistency checks can be done in parallel, ensuring that $\mathcal{C}$ has polylogarithmic depth.

How to sort with a non-deterministic circuit. A *routing network* is a graph with a designated set of T source vertices and a designated set of T sink vertices (both sets of the same cardinality) satisfying the following property: for any perfect matching between sources and sinks (equivalently, for any desired sorting of the sources), there is a set of node-disjoint paths that connects each source to the sink to which it is matched. The specific routing network used in $\mathcal{C}$ is derived from a *De Bruijn* graph G. G consists of $\ell = O(\log T)$, with T nodes at each layer. The first layer consists of the source vertices, and the last layer consists of the sinks. Each node at intermediate layers has exactly two in-neighbors and exactly two out-neighbors.

The precise definition of the De Bruijn graph G is not essential to the discussion here. What is important is that G satisfies the following two properties.

Property 1: Given any desired sorting of the sources, a corresponding routing can be found in $O(|G|) = O(T \cdot \log T)$ time using known routing algorithms [Ben65, Wak68, Lei92, BSCGT13].

Property 2: The multilinear extension of the wiring predicate of G can be evaluated in polylogarithmic time. By wiring predicate of G, we mean the Boolean function (analogous to the functions add_i and mult_i in the GKR protocol) that takes as input the labels $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ of three nodes in G, and outputs 1 if and only if $\mathbf{b}$ and $\mathbf{c}$ are the in-neighbors of $\mathbf{a}$ in G.

Roughly speaking, Property 2 holds because in a De Bruijn graph, the neighbors of a node with label **v** are obtained from **v** by simple bit shifts, which is a "degree-1 operation" in the following sense. The function that tests whether two binary labels are bit-shifts of each other is an AND of pairwise disjoint bit equality tests. The direct arithmetization of such a function (replacing the AND gate with multiplication, and the bitwise equality tests with their multilinear extensions) is multilinear.

In a routing of G, each node **v** other than source nodes has a single in-neighbor in the routing (we think of this in-neighbor as forwarding its packet to **v**), and each node **v** other than sink nodes has exactly one out-neighbor in the routing. Thus, a routing in G can be specified by assigning each non-sink node **v** a single bit that specifies which of **v**'s two out-neighbors in G get forwarded a packet by **v**. To perform the sorting step, the circuit will take additional bits as non-deterministic input, called *routing bits*, which give the bit-wise specification of a routing just described.

To put everything together, the circuit $\mathcal{C}$ sorts the (timestamps, list) pairs of the transcript from time-order into memory order by implementing the routing network G as follows. For each node **v** in G, $\mathcal{C}$ contains a "gadget" of polylog$n$ gates. The gadget for v takes as input a (timestamp, list) pair, which is viewed as a packet, as well as the routing bit $b_{\mathbf{v}}$ for **v**. Based on $b_{\mathbf{v}}$ it "forwards" its packet to the appropriate out-neighbor of **v**.

The Wiring Predicates of $\mathcal{C}$. The circuit $\mathcal{C}$ has a very regular wiring structure, with lots of repeated structure. Specifically, its time-consistency-checking circuitry applies the same small sub-circuit capturing independently to every time step in the witness, and (after resorting the witness into memory order), the memory-consistency-checking circuitry is also data parallel.

All told, it is possible to exploit this data parallel structure (plus meanwhile, Property 2 of G above, which ensures that the sorting circuitry also has a nice, regular wiring structure) to show that (a slight modification of) the multilinear extensions $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ of $\mathcal{C}$ can be evaluated in polylogarithmic time.

This ensures that if one applies the GKR protocol (in combination with a commitment scheme as described in Section 1) that the verifier can run in time $O(n + \text{polylog}(T))$, without ever having to look at the gates of $\mathcal{C}$ individually. Moreover, the prover can generate the entire circuit $\mathcal{C}$ and the witness w, and perform its part of the GKR protocol applied to $\mathcal{C}(x, w)$ in time $O(T \cdot \text{polylog}(T))$.

Intuitively, in the resulting argument system, the verifier is forcing the prover not only to run the RAM M on input x, but also to produce a transcript of the execution and then *confirm* via the circuit $\mathcal{C}$ that the transcript contains no errors. Fortunately, it does not require much more work for the prover to produce the transcript and confirm its correctness then it does to run M on x in the first place.

References

[Ben65] Vaclav E. Beneš. Mathematical theory of connecting networks and telephone traffic. *Academic Press*, 1965.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS*, pages 401–414. ACM, 2013.

[Lei92] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., 1992.

[Wak68] Abraham Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.