COSC 544 Probabilistic Proof Systems 10/3/17

Turning Computer Programs Into Circuits (Part 1)

Lecturer: Justin Thaler

1 Introduction

In the previous lecture, we saw a very efficient interactive proof, called the GKR protocol, for verifiably outsourcing the evaluation of large arithmetic circuits, as long as the circuit is not too deep. But in the real world, people are rarely interested in evaluating giant arithmetic circuits. Rather, they typically have a computer program written in a high-level programming language like Java or Python, and want to execute the program on their data. In order for the GKR protocol to be useful in this setting, we need an efficient way to turn high-level computer programs into arithmetic circuits. We can then apply the GKR protocol to the resulting arithmetic circuit.

Some computer programs naturally lend themselves to implementation via arithmetic circuits, particularly programs that only involve addition and multiplication of integers or elements of a fine field. For example, the following circuit implements the standard naive $O(n^3)$ time algorithm for multiplying two $n \times n$ matrices, A and B.

Let $[n] := \{1, \dots, n\}$. Adjacent to the input layer of the circuit is a layer of n^3 multiplication gates, each assigned a label $(i, j, k) \in [n] \times [n] \times [n]$. Gate (i, j, k) at this layer computes the product of $A_{i,k}$ and $B_{k,j}$. Beneath this layer of multiplication gates lies a binary tree of addition gates of depth $\log_2(n)$. This ensures that there are n^2 output gates, and if we assign each output gate a label $(i, j) \in [n] \times [n]$, then the (i, j)'th output gate computes $\sum_{k \in [n]} A_{i,k} \cdot B_{k,j}$ as required by the definition of matrix multiplication. See Figure 1.

Remark 1. One can obtain an interactive proof for matrix multiplication by applying the GKR protocol to the observe circuit. This circuit exhibits a nice, regular wiring pattern that allows all of the optimizations from the previous lecture to apply, resulting in a prover that runs in time linear in the size of the circuit, and a verifier that runs in time linear in the size of the input. But because the circuit has size $\Omega(n^3)$, this is still much worse for the prover than the special-purpose matrix multiplication protocol of Lecture 7.

Effectively, applying the GKR protocol to the circuit above *forces* the prover to use naive the matrix multiplication algorithm to compute the correct answer $A \cdot B$. The special purpose matrix multiplication protocol of Lecture 7 allowed the prover to find the correct answer any way it wanted, and then do a low-order amount of extra work to prove that the answer is correct.

While it is fairly straightforward to turn the algorithm for naive matrix multiplication into an arithmetic circuit as above, other kinds of computer programs that perform "non-arithmetic" operations (such as evaluating complicated conditional statements) seem to be much more difficult to turn into small arithmetic circuits.

In this lecture, we will see two techniques for turning arbitrary computer programs into circuits. Next lecture, we will see a third technique, which is far more practical, but makes makes use of what are called "non-deterministic circuits".

We would like to make statements like "Any computer programming that halts within $T(n)$ time steps can be turned into a (low-depth, layered, fan-in two) arithmetic circuit of size at most $T(n) \log T(n)$." In order to make statements of this form, we first have to be precise about what it means to say that a computer programs has runtime T.

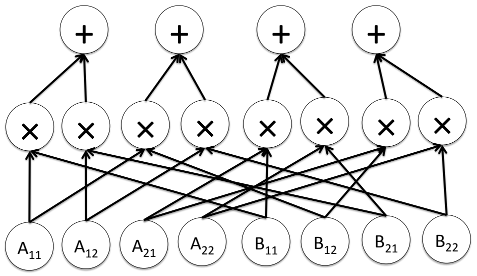

Figure 1: An Arithmetic Circuit Implementing The Naive Matrix Multiplication Algorithm for 2×2 matrices.

2 Machine Code

Modern compilers are very good at efficiently turning high-level computer programs into *machine code*, which is a set of basic instructions that can each be executed in unit time by on the machine's hardware. When we say that a program runs in $T(n)$ time steps, we mean that it can be compiled into a sequence of machine instructions of length at most $T(n)$. But for this statement to be precise, we have to decide precisely what is a machine instruction. That is, we have to specify (a model of) the hardware on which we will think of our programs as running.

Our hardware model will be a simple *random access machine* (RAM). A RAM consists of the following components.

- (Main) Memory. That is, it will contain s cells of storage, where each cell can store, say, 64 bits of data.
- A constant number (say, 8) of registers. Registers are special memory cells with which the RAM can manipulate data. That is, whereas Main Memory cells can only store data, the RAM is allowed to perform operations on data in registers, such as "add the numbers in Registers 1 and 2, and store the result in Register 3".
- A set of $\ell = O(1)$ allowed machine instructions. Typically, these instructions are of the form:
  - Write the value currently stored in a given register to a specific location in Main Memory.
  - Read the value from a specific location in Main Memory into a register.
  - Perform basic manipulations of data in registers. For example, adding, subtracting, multiplying, dividing, or comparing the values stored in two registers, and storing the result in a third register. Or doing bitwise operations on the values stored in two registers (i.e., computing the bit-wise AND of two values). Etc.
  - A program counter. This is a special register that tells the machine what is the next instruction to execute.

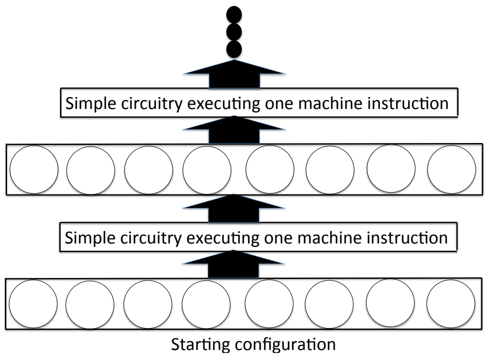
Simple circuitry executing one machine instruction
Simple circuitry executing one machine instruction
Starting configuration

Figure 2: A caricature of a technique for turning any program running in time $T(n)$ and space $S(n)$ into a circuit of depth not much more than $T(n)$ and width not much more than $s(n)$.

3 A First Technique For Turning Programs Into Circuits [Sketch]

Our first technique for turning computer programs into circuits yields the following. If a computer programs runs in time $T(n)$ on a RAM with at most $s(n)$ cells of memory, then the program can be turned into a (layered, fan-in 2) arithmetic circuit of depth not much more than $T(n)$ and *width* of about $s(n)$ (i.e., the number of gates at each layer of the circuit is not much more than $s(n)$).

Observe that such a transformation from programs to circuits is useless in the context of the GKR protocol, because the verifier's time complexity in the GKR protocol is at least the circuit depth, which is about $T(n)$ in this construction. In time $T(n)$, the verifier could have executed the entire program on her own, without any help from the prover. We describe this circuit-generation technique even though it is useless in the context of the GKR protocol because it is conceptually important.

This program-to-circuit transformation makes use of the notion of a machine *configuration*. A machine configuration tells you everything about the state of a RAM at a given time. That is, it specifies the input, as well as the value of every single memory cell, register, and program counter. Observe that if if a RAM has a memory of size s, then a configuration can be specified with roughly $64s$ bits (where 64 is the number of bits that can fit in one memory cell), plus some extra bits to specify the input and the values stored in the registers and program counter.

The basic idea of the transformation is to have the circuit proceed in iterations, one for each time step of the computer program. The ith iteration takes as input the configuration of the RAM after i steps of the program have been executed, and "executes one more step of the program". That is, it determines what the configuration of the RAM would be after the $(i+1)$'st machine instruction is executed. This is displayed pictorially in Figure 2.

A key point that makes this transformation work is that there is only a constant number of possible machine instructions, each of which is very simple (operating on only a constant number of registers, in a simple manner). Hence, the circuitry that maps the configuration of the machine after i steps of the program to the configuration after the $(i+1)$'st step is very simple.

4 Turning Small-Space Programs Into Shallow Circuits

The circuits that come out of the program-to-circuit transformation of the previous section are useless in the context of the GKR protocol because the circuits have depth at least T. When applying the GKR protocol to such a deep circuit, the runtime of the verifier is at least T. It would be just as fast for the verifier to simply run the computer program on its own, without bothering with a prover.

Our second technique for turning computer programs generates shallower circuits as long as the computer program doesn't use too much space. Specifically, it is capable of taking any program that runs in time T and space s, and turning it into a circuit of depth roughly $s \cdot \log T$ and size $2^{\Theta(s)}$.

This technique makes use of the notion of the *configuration graph* of a random access machine M on input x. The configuration graph has a vertex for every possible configuration of M—observe that for any fixed input x, there are $2^{\Theta(s)}$ possible configurations of M, where s is the number of bits in M's main memory. This is because s bits can take on 2^s different values. There is a directed edge in the configuration graph connecting configuration A to configuration B if, when starting in configuration A, running M for one step results in configuration B.

Let us assume for simplicity that M computes a decision problem, i.e., it outputs either 0 or 1, and let us insist (still for simplicity) that M reset all of its memory and registers to 0 upon termination, except for a designated register meant to store the output. Then there are only two possible output configurations of M; one in which the output register stores 1, and all other registers and memory cells are set to 0, and one in which the output register stores 0 and all other registers and memory cells are set to 0. Let us call the former output configuration the "accepting configuration".

Hence, we can determine whether M outputs 1 on input x in less than T time steps by determining whether there is a directed path of length at most T in M's configuration graph from the starting configuration to the accepting configuration. Hence, we have reduced the task of running M for T steps to a reachability problem in a directed graph.

Essentially, our circuit will implement an efficient parallel algorithm for solving directed reachability in the configuration graph of M. In more detail, let A be the adjacency matrix of M's configuration graph; that is, $A_{i,j} = 1$ if configuration i has a directed edge to configuration j in M's configuration graph, and $A_{i,j} = 0$ otherwise.

Observe that $A^2_{i,j} > 0$ if and only if there is a directed path of length at most 2 from configuration i to configuration j. Similarly, $A^4 > 0$ if and only if there is a directed path of length at most 4 from configuration i to configuration j. And so on.

Hence, in order to determine whether there is a directed path of length at most T from the starting configuration to the accepting configuration, it is enough for the circuit to repeatedly square the adjacency matrix $\log T$ times. We have seen in Section 1 that there is a circuit of size $O(N^3)$ and depth $O(\log N)$ for multiplying two $N \times N$ matrices. Since the configuration graph of M on input x is an $2^{\Theta(s)} \times 2^{\Theta(s)}$ matrix, our circuit that squares the adjacency matrix of the configuration graph of M $O(\log T)$ times has depth $O(\log(2^{\Theta(s)}) \cdot \log T) = O(s \log T)$, and size $2^{\Theta(s)}$.

4.1 IP = PSPACE As a Consequence of the GKR Protocol

Recall that the verifier's runtime in the GKR protocol when applied to a circuit $\mathcal{C}$ is $O(n + d \log S + t)$, where d is the depth of $\mathcal{C}$, S is the size, and t is the time required to evaluate the multilinear extensions of the wiring predicates add_i and mult_i at each layer i of $\mathcal{C}$ at a point.

For the circuit of depth $O(s \log T)$ and size $2^{\Theta(s)}$ described above, $\widetilde{\text{add}}_i$ and $\widetilde{\text{mult}}_i$ can be evaluated in time polynomial in n and s. We omit the tedious details that must be dealt with to establish this statement, but roughly speaking, it holds because (a) the configuration graph of a random access machine is highly structured, and (b) matrix multiplication can be implemented by a circuit with a highly regular wiring pattern (as we saw in Section 1).

Hence, by applying the GKR protocol to the circuit described above, we obtain an interactive proof for any problem solving in space s on inputs of size n, in which the verifier runs in time polynomial in n and s. An immediate consequence is that $\mathbf{IP} = \mathbf{PSPACE}$.

Remark 2. Recall that the circuit generated by the program-to-circuit compilation procedure above has size at least 2^s, which is super polynomial as long as $s = \omega(\log n)$. Since the prover in the GKR protocol has to evaluate the circuit gate-by-gate, this means that the when applying the GKR protocol to this circuit, the prover runs in superpolynomial time whenever $s = \omega(\log n)$, even if the original program itself only a polynomial number of steps. Effectively, the circuit winds up exploring *all possible configurations* of M, even only a tiny fraction of all possible configurations are ever reached when M is actually run on input x. By using this circuit within the GKR protocol, the prover is itself forced to explore all of these unused configurations.

A recent breakthrough of Reingold, Rothblum, and Rothblum [RRR16] addresses this limitation. Specifically [RRR16] gives an interactive proof with a polynomial-time prover and a nearly-linear time verifier for any problem solvable in space roughly $O(n^{1/2})$.

References

[RRR16] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 49–62, New York, NY, USA, 2016. ACM.