Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to
complexitybook@gmail.com — **Thank You!!**

Chapter 14

Algebraic computation models

The Turing machine model captures computations on bits (equivalently, integers), but it does not always capture the spirit of algorithms which operate on, say the real numbers $\mathbb{R}$ or complex numbers $\mathbf{C}$. Such algorithms arise in a variety of applications such as numerical analysis, computational geometry, robotics, and symbolic algebra. A simple example is *Newton's method* for finding roots of a given real-valued function function f. It iteratively produces a sequence of candidate solutions $x_0, x_1, x_2, \ldots, \in \mathbb{R}$ where $x_{i+1} = x_i - f(x_i)/f'(x_i)$. Under appropriate conditions this sequence can be shown to converge to a root of f.

Of course, a perfectly defensible position to take is that even the behavior of such algorithms should be studied using TMs, since they will be run on real-life computers, which represent real numbers using finite precision. In this chapter though, we take a different approach and study models which do allow arithmetic operations on real numbers (or numbers from fields other than $\mathbb{R}$). Such an idealized model may not be implementable, strictly speaking, but it provides a useful approximation to the asymptotic behavior as computers are allowed to use more and more precision in their computations. Furthermore, one may be able to prove nontrivial lowerbounds for these models using techniques from well-developed areas of mathematics such as algebraic geometry and topology. (By contrast, boolean circuit lowerbounds have proven very difficult.)

However, coming up with a meaningful, well-behaved model of algebraic computation is not an easy task, as the following example suggests.

Example 14.1 (Pitfalls awaiting designers of such models)
A real number can encode infinite amount of information. For example, a single real number is enough to encode the answer to every instance of SAT (or any other language, in general). Thus, a model that can store any real number with infinite precision may not be realistic. Shamir has shown how to factor any integer n in $\text{poly}(\log n)$ time on a computer that can do real arithmetic with arbitrary precision.

The usual way to avoid this pitfall is to restrict the algorithms' ability to access individual bits (e.g., the machine may require more than polynomial time to extract a particular digit from

a real number). Or, sometimes (as in case of Algebraic Computation Trees) it is OK to consider unrealistically powerful models since the goal is to prove nontrivial lowerbounds —say, superlinear or quadratic— rather than arbitrary polynomial lowerbounds. After all, lowerbounds for unrealistically powerful models will apply to more realistic (and weaker) models as well.

This chapter is a sketchy introduction to algebraic complexity. It introduces three algebraic computation models: algebraic circuits, algebraic computation trees, and algebraic Turing Machines. The algebraic TM is closely related to the standard Turing Machine model and allows us to study similar questions for arbitrary fields — including decidability and complexity–that we earlier studied for strings over $\{0,1\}$. We introduce an undecidable problem (namely, deciding membership in the Mandelbrot set) and and an **NP**-complete problem (decision version of Hilbert's Nullstellensatz) in this model.

14.1 Algebraic circuits

An *algebraic circuit* over a field F is defined by analogy with a boolean circuit. It consists of a directed acyclic graph. The leaves are called input nodes and labeled $x_1, x_2, \ldots, x_n$, except these take values in a field F rather than boolean variables. There are also special input nodes, labeled with the constants 1 and -1 (which are field elements). Each internal node, called a *gate*, is labeled with one of the arithmetic operations $\{+, \star\}$ rather than with the boolean operations $\vee, \wedge, \neg$ used in boolean circuits. There is only output node. We restrict indegree of each gate to 2. The *size* of the circuit is the number of gates in it. One can also consider algebraic circuits that allow division ($\div$) at the gates. One can also study circuits that have access to "constants" other than 1; though typically one assumes that this set is fixed and independent of the input size n. Finally, as in the boolean case, if each gate has outdegree 1, we call it an *arithmetic formula*.

A gate's operation consists of performing the operation it is labeled with on the numbers present on the incoming wires, and then passing this output to all its outgoing wires. After each gate has performed its operation, an output appears on the circuit's lone output node. Thus the circuit may be viewed as a computing a function $f(x_1, x_2, \ldots, x_n)$ of the input variables, and simple induction shows that this output function is a (multivariate) polynomial in $x_1, x_2, \ldots, x_n$. If we allow gates to also be labeled with the division operation (denoted "$\div$") then the function is a rational function of $x_1, \ldots, x_n$, in other words, functions of the type $f_1(x_1, x_2, \ldots, x_n)/f_2(x_1, \ldots, x_n)$ where f_1, f_2 are polynomials. Of course, if the inputs come from a field such as $\mathbb{R}$, then rational functions can be used to approximate —via Taylor series expansion —all "smooth" real-valued functions.

As usual, we are interested in the asymptotic size (as a function of n) of the smallest family of algebraic circuits that computes a family of polynomials $\{f_n\}$ where f_n is a polynomial in n variables. The exercises ask you to show that circuits over $GF(2)$ (with no $\div$) are equivalent to boolean circuits, and the same is true for circuits over any finite field. So the case when F is infinite is usually of greatest interest.

EXAMPLE 14.2
The *discrete fourier transform* of a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ where $a_i \in \mathbf{C}$ is vector $M \cdot \mathbf{a}$, where M is a fixed $n \times n$ matrix whose (i,j) entry is ω^{ij} where ω is an nth root of 1 (in other words, a complex number satisfying $\omega^n = 1$).

Interpreting the trivial algorithm for matrix-vector product as an arithmetic circuit, one obtains an algebraic formula of size $O(n^2)$. Using the famous *fast fourier transform* algorithm, one can obtain a smaller circuit (or formula??; CHECK) of size $O(n \log n)$.

STATUS OF LOWERBOUNDS??

EXAMPLE 14.3
The *determinant* of an $n \times n$ matrix $X = (X_{ij})$ is

$$det(X) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i\sigma(i)}, \tag{1}$$

where S_n is the set of all $n!$ permutations on $\{1, 2, \ldots, n\}$. This can be computed using the familiar Gaussian elimination algorithm. Interpreting the algorithm as a circuit one obtains an arithmetic circuit of size $O(n^3)$. Using the $\mathbf{NC}^2$ algorithm for Gaussian elimination, one obtains an arithmetic formula of size $2^{O(\log^2 n)}$. No matching lowerbounds are known for either upperbound.

The previous example is a good illustration of how the polynomial defining a function may have exponentially many terms —in this case $n!$—but nevertheless be computable with a polynomial-size circuit (as well as a subexponential-size formula).

By contrast, no polynomial-size algebraic circuit is conjectured to exist for the *permanent* function, which at first sight seems is very similar to the determinant but as we saw in Section **??**, is #**P**-complete.

$$permanent(X) = \sum_{\sigma \in S_n} (-1)^{sgn(\sigma)} \prod_{i=1}^{n} x_{i\sigma(i)}, \tag{2}$$

The determinant and permanent functions also play a vital role in the world of algebraic circuits, since they are *complete* problems for two important classes. To give the definition, we need the notion of *degree* of a multivariate polynomial, namely, the minimum d such that each monomial term $\prod_i x_i^{d_i}$ satisfies $\sum_i d_i \leq d$. A family of polynomials in $x_1, x_2, \ldots, x_n$ is *poly-bounded* if the degree is at most $O(n^c)$ for some constant $c > 0$.

DEFINITION 14.4 ($Alg\mathbf{P}$)
The class $Alg\mathbf{P}$ is the class of polynomials of polynomial degree that are computable by arithmetic formulae (using no $\div$) of polynomial size.

DEFINITION 14.5 (ALG**NP**)
Alg**NP** is the class of polynomials of polynomial degree that are definable as

$$f(x_1, x_2, \ldots, x_n) = \sum_{e \in \{0,1\}^{m-n}} g_n(x_1, x_2, \ldots, x_n, e_{n+1}, \ldots, e_m),$$

where $g_n \in Alg\mathbf{P}$ and m is polynomial in n.

Definition 14.6 (Projection reduction)
A function $f(x_1, \ldots, x_n)$ is a *projection* of a function $g(y_1, y_2, \ldots, y_m)$ if there is a mapping σ from $\{y_1, y_2, \ldots, y_m\}$ to $\{0, 1, x_1, x_2, \ldots, x_n\}$ such that $f(x_1, x_2, \ldots, x_n) = g(\sigma(y_1), \sigma(y_2), \ldots, \sigma(y_m))$.

We say that f is *projection-reducible* to g if f is a projection of g.

Theorem 14.7 (Valiant)
Every polynomial on n variables that is computable by a circuit of size u is projection reducible to the Determinant function (over the same field) on $u + 2$ variables.

*Every function in Alg***NP** *is projection reducible to the Permanent function (over the same field).*

14.2 Algebraic Computation Trees

An algebraic computation tree is reminiscent of a boolean decision tree (Chapter **??**) but it computes a boolean-valued function $f : \mathbb{R}^n \to \{0, 1\}$. Consider for example the ELEMENT DISTINCTNESS problem of deciding, given n numbers $x_1, x_2, \ldots, x_n$, whether any two of them are the same. To study it in the decision tree model, we might study it by thinking of the input as a matrix of size n^2 where the (i, j) entry indicates whether or or not $x_i > x_j$ or $x_i = x_j$ or $x_i < x_j$. But one can also study it as a problem whose input is a vector of n real numbers. Consider the trivial algorithm in either viewpoint: sort the numbers in $O(n \log n)$ time and then check if any two adjacent numbers in the sorted order are the same. Is this trivial algorithm actually optimal? This question is still open, but one can prove optimality with respect to a more restricted class of algorithms that includes the above trivial algorithm.

Recall that comparison-based sorting algorithms only ask questions of the type "Is $x_i > x_j$?", which is the same as asking whether $x_i - x_j > 0$. The left hand side term of this last inequality is a linear function. Other algorithms may use more complicated functions. In this section we consider a model called *Algebraic Computation Trees*, where we examine the effect of allowing a) the use of any polynomial function and b) the introduction of new variables together with the ability to ask questions about them.

Definition 14.8 (Algebraic Computation Tree)
An *Algebraic Computation Tree* is a way to represent a function $f : \Re^n \to \{0, 1\}$ by showing how to compute $f(x_1, x_2, \ldots, x_n)$ for any input vector $(x_1, x_2, \ldots, x_n)$. It is a complete binary tree that describes where each of the nodes has one of the following types:

- Leaf labeled "Accept" or "Reject".
- Computation node v labeled with y_v, where $y_v = y_u \circ y_w$ and y_u, y_w are either one of $\{x_1, x_2, \ldots, x_n\}$ or the labels of ancestor nodes and the operator $\circ$ is in $\{+, -, \times, \div, \sqrt{}\}$.
- Branch node with out-degree 2. The branch that is taken depends on the evaluation of some condition of the type $y_u = 0$ or $y_u \geq 0$ or $y_u \leq 0$ where y_u is either one of $\{x_1, x_2, \ldots, x_n\}$ or the labels of an ancestor node in the tree.

Figure unavailable in pdf file.

Figure 14.1: An Algebraic Computation Tree

Figure unavailable in pdf file.

Figure 14.2: A computation path p of length d defines a set of constraints over the n input variables x_i and d additional variables y_j, which correspond to the nodes on p.

The computation on any input $(x_1, x_2, \ldots, x_n)$ follows a single path the root to a leaf, evaluating functions at internal nodes (including branch nodes) in the obvious way. The complexity of the computation on the path is measured using the following costs (which reflect real-life costs to some degree):

- $+, -$ are free.
- $\times, \div, \sqrt{}$ are charged unit cost.

The *depth* of the tree is the maximum cost of any path in it.

A fragment of an algebraic decision tree is shown in figure 14.1. The following examples illustrate some of the languages (over real numbers) whose complexity we want to study.

EXAMPLE 14.9
[Element Distinctness Problem] Given n numbers $x_1, x_2, \ldots, x_n$ we need to determine whether they are all distinct. This is equivalent to the question whether $\prod_{i \neq j}(x_i - x_j) \neq 0$. As indicated earlier, this can be computed by a tree of depth $O(n \log n)$ whose internal nodes only compute functions of the type $x_i - x_j$.

EXAMPLE 14.10
[Real number version of SUBSET SUM] Given a set of n real numbers $X = \{x_1, x_2, \ldots, x_n\}$ we ask whether there is a subset $S \subseteq X$ such that $\sum_{i \in S} x_i = 1$.

Of course, a tree of depth d could have 2^d nodes, so a small depth decision tree does not always guarantee an efficient algorithm. This is why the following theorem (which we do not prove) does not have any implication for **P** versus **NP**.

THEOREM 14.11
The real number version of SUBSET SUM *can be solved using an algebraic computation tree of depth* $O(n^5)$.

This theorem suggests that Algebraic Computation Trees are best used to investigate lowerbounds such as $n \log n$ or n^2. To prove lowerbounds for a function f, we will use the *topology* of the sets $f^{-1}(1)$ and $f^{-1}(0)$, specifically, the number of connected components. In fact, we will think of any function $f : \mathbb{R}^n \to \mathbb{R}$ as being defined by a subset $W \subseteq \mathbb{R}^n$, where $W = f^{-1}(1)$.

Definition 14.12
Let $W \subseteq \mathbb{R}^n$. The *algebraic computation tree complexity* of W is

$$C(W) = \min_{\substack{\text{computation} \\ \text{trees } C \text{ for } W}} \{\text{depth of } C\}$$

Definition 14.13 (connected components)
A set $S \subseteq \mathbb{R}^n$ is *connected* if for all $x, y \in S$ there is path p that connects x and y and lies entirely in S. For $S \subseteq \mathbb{R}^n$ we define $\#(S)$ to be the number of connected components of S.

Theorem 14.14
Let $W = \{(x_1, \ldots, x_n) | \prod_{i \neq j} (x_i - x_j) \neq 0\}$. *Then,*

$$\#(W) \geq n!$$

Proof: For each permutation σ let

$$W_\sigma = \{(x_1, \ldots, x_n) \mid x_{\sigma(1)} < x_{\sigma(2)} < \ldots < x_{\sigma(n)}\}.$$

That is, let W_σ be the set of n-tuples $(x_1, \ldots, x_n)$ to which σ gives order. It suffices to prove for all $\sigma' \neq \sigma$ that the sets W_σ and $W_{\sigma'}$ are not connected.

For any two distinct permutations σ and σ', there exist two distinct i, j with $1 \leq i, j \leq n$, such that $\sigma^{-1}(i) < \sigma^{-1}(j)$ but $\sigma^{-1}(i) > \sigma^{-1}(j)$. Thus, in W_σ we have $X_j - X_i > 0$ while in $W_{\sigma'}$ we have $X_i - X_j > 0$. Consider any path from W_σ to $W_{\sigma'}$. Since $X_j - X_i$ has different signs at the endpoints, the intermediate value principle says that somewhere along the path this term must become 0. Definition 14.13 then implies that W_σ and $W_{\sigma'}$ cannot be connected. ■

The connection between the two parameters we have defined thus far is the following theorem, whose proof will use a fundamental theorem of topology. It also implies, using our observation above, that the algebraic computation tree complexity of ELEMENT DISTINCTNESS is $\Omega(\log(n!)) = \Omega(n \log n)$.

Theorem 14.15 (Ben-Or)

$$C(W) = \Omega\Big(\log \left(\max \{\#(W), \#(\mathbb{R}^n - W)\} \right) - n \Big)$$

This theorem is proved in two steps. First, we try to identify the property of functions with low decision tree complexity: they can be defined using a "few" systems of equations.

LEMMA 14.16
If $f:\Re^n \to \{0,1\}$ has a decision tree of depth d then $f^{-1}(1)$ (and also $f^{-1}(0)$) is a union of at most 2^d sets $C_1, C_2, \ldots$, where C_i is the set of solutions to some algebraic system of up to d equations of the type

$$p_i(y_1, \ldots, y_d, x_1, \ldots, x_n) \bowtie 0,$$

where p_i for $i \leq d$ is a degree 2 polynomial, $\bowtie$ is in $\{\leq, \geq, =, \neq\}$, and $y_1, \ldots, y_d$ are new variables.

(Rabinovitch's Trick) Additionally, we may assume without loss of generality (at the cost of doubling the number of y_i's) that there are no $\neq$ constraints in this system of equations.

PROOF: The tree has 2^d leaves, so it suffices to associate a set with each leaf. This is simply the set of $(x_1, x_2, \ldots, x_n)$ that end up at that leaf. Associate variables $y_1, y_2, \ldots, y_d$ with the d tree nodes appearing along the path from root to that leaf. For each tree nodes associate an equation with it in the obvious way (see figure 14.2). For example, if the node computes $y_v = y_u \div y_w$ then it implies the constraint $y_v y_w - y_u = 0$. Thus any $(x_1, x_2, \ldots, x_n)$ that end up at the leaf is a vector with an associated value of $y_1, y_2, \ldots, y_d$ such that the combined vector is a solution to these d equations.

To replace the "$\neq$" constraints with "$=$" constraints we take a constraint like

$$p_i(y_1, \ldots, y_m) \neq 0,$$

introduce a new variable z_i and impose the constraint

$$q_i(y_1, \ldots, y_m, z_i) \equiv 1 - z_i p_i(y_1, \ldots, y_m) = 0.$$

(This transformation holds for all fields.) Notice, the maximum degree of the constraint remains 2, because the trick is used only for the branch $y_u \neq 0$ which is converted to $1 - z_v y_u = 0$.

■

REMARK 14.17
We find Rabinovitch's trick useful also in Section 14.3.2 where we prove a completeness result for Hilbert's Nullstellensatz.

Another version of the trick is to add the constraint

$$p_i^2(y_1, \ldots, y_m) > 0,$$

which doubles the degree and does not hold for all fields (e.g., the complex numbers).

Thus we need some result about the number of connected components of the set of solutions to an algebraic system. The following is a central result in mathematics.

THEOREM 14.18 (SIMPLE CONSEQUENCE OF MILNOR-THOM)
If $S \subseteq \mathbb{R}^n$ is defined by degree d constraints with m equalities and h inequalities then

$$\#(S) \leq d(2d-1)^{n+h-1}$$

REMARK 14.19
Note that the above upperbound is independent of m.

Figure unavailable in pdf file.

Figure 14.3: Projection can merge but not add connected components

Now we can prove Ben-Or's Theorem.

PROOF: (Theorem 14.15) Suppose that the depth of a computation tree for W is d, so that there are at most 2^d leaves. We will use the fact that if $S \subseteq \mathbb{R}^n$ and $S|_k$ is the set of points in S with their $n-k$ coordinates removed (projection on the first k coordinates) then $\#(S|_k) \leq \#(S)$ (figure 14.3).

For every leaf there is a set of degree 2 constraints. So, consider a leaf ℓ and the corresponding constraints $\mathcal{C}_\ell$, which are in variables $x_1, \ldots, x_n, y_1, \ldots, y_d$. Let $W_\ell \subseteq \mathbb{R}^n$ be the subset of inputs that reach ℓ and $S_\ell \subseteq \mathbb{R}^{n+d}$ the set of points that satisfy the constraints $\mathcal{C}_\ell$. Note that $W_\ell = \mathcal{C}_\ell|_n$ i.e., W_ℓ is the projection of $\mathcal{C}_\ell$ onto the first n coordinates. So, the number of connected components in W_ℓ is upperbounded by $\#(\mathcal{C}_\ell)$. By Theorem 14.18 $\#(\mathcal{C}_\ell) \leq 2 \cdot 3^{n+d-1} \leq 3^{n+d}$. Therefore the total number of connected components is at most $2^d 3^{n+d}$, so $d \geq \log(\#(W)) - O(n)$. By repeating the same argument for $\mathbb{R}^n - W$ we have that $d \geq \log(\#(\mathbb{R}^n - W)) - O(n)$. ■

14.3 The Blum-Shub-Smale Model

Blum, Shub and Smale introduced Turing Machines that compute over some arbitrary field K (e.g., $K = \mathbb{R}, \mathbf{C}, \mathbb{Z}_2$). This is a generalization of the standard Turing Machine model which operates over the ring $\mathbb{Z}_2$. Each cell can hold an element of K, Initially, all but a finite number of cells are "blank." In our standard model of the TM, the computation and branch operations can be executed in the same step. Here we perform these operations separately. So we divide the set of states into the following three categories:

- Shift state: move the head to the left or to the right of the current position.
- Branch state: if the content of the current cell is a then goto state q_1 else goto state q_2.
- Computation state: replace the contents of the current cell with a new value. The machine has a hardwired function f and the new contents of the cell become $a \leftarrow f(a)$. In the standard model for rings, f is a polynomial over K, while for fields f is a rational function p/q where p, q are polynomials in $K[x]$ and $q \neq 0$. In either case, f can be represented using a constant number of elements of K.
- The machine has a single "register" onto which it can copy the contents of the cell currently under the head. This register's contents can be used in the computation.

In the next section we define some complexity classes related to the BSS model. As usual, the time and space complexity of these Turing Machines is defined with respect to the input size, which is the number of cells occupied by the input.

REMARK 14.20
The following examples show that some modifications of the BSS model can increase significantly the power of an algebraic Turing Machine.

- If we allow the branch states to check, for arbitrary real number a, whether $a > 0$ (in other words, with arbitrary precision) the model becomes unrealistic because it can decide problems that are undecidable on the normal Turing machine. In particular, such a machine can compute $\mathbf{P}/poly$ in polynomial time; see Exercises. (Recall that we showed that $\mathbf{P}/poly$ contains undecidable languages.) If a language is in $\mathbf{P}/poly$ we can represent its circuit family by a single real number hardwired into the Turing machine (specifically, as the coefficient of of some polynomial $p(x)$ belonging to a state). The individual bits of this coefficient can be accessed by dividing by 2, so the machine can extract the polynomial length encoding of each circuit. Without this ability we can prove that the individual bits cannot be accessed.
- If we allow rounding (computation of $\lfloor x \rfloor$) then it is possible to factor integers in polynomial time, using some ideas of Shamir. (See exercises.)

Even without these modifications, the BSS model seems more powerful than real-world computers: Consider the execution of the operation $x \leftarrow x^2$ for n times. Since we allow each cell to store a real number, the Turing machine can compute and store in one cell (without overflow) the number x^{2^n} in n steps.

14.3.1 Complexity Classes over the Complex Numbers

Now we define the corresponding to $\mathbf{P}$ and $\mathbf{NP}$ complexity classes over $\mathbf{C}$:

DEFINITION 14.21 ($\mathbf{P_C}$,$\mathbf{NP_C}$)
$\mathbf{P_C}$ is the set of languages that can be decided by a Turing Machine over $\mathbf{C}$ in polynomial time. $\mathbf{NP_C}$ is the set of languages L for which there exists a language L_0 in $\mathbf{P_C}$, such that an input x is in L iff there exists a string $(y_1, \ldots, y_{n^c})$ in $\mathbf{C}^{n^c}$ such that (x, y) is in L_0.

The following definition is a restriction on the inputs of a TM over $\mathbf{C}$. These classes are useful because they help us understand the relation between algebraic and binary complexity classes.

DEFINITION 14.22 (0-1-$\mathbf{NP_C}$)

$$\text{0-1-}\mathbf{NP_C} = \{L \cap \{0,1\}^* \mid L \in \mathbf{NP_C}\}$$

Note that the input for an $\mathbf{NP_C}$ machine is binary but the nondeterministic "witness" may consist of complex numbers. Trivially, 3SAT is in 0-1-$\mathbf{NP}_C$: even though the "witness" consists of a string of complex numbers, the machine first checks if they are all 0 or 1 using equality checks. Having verified that the guess represents a boolean assignment to the variables, the machine continues as a normal Turing Machine to verify that the assignment satisfies the formula.

It is known that 0-1-$\mathbf{NP_C} \subseteq \mathbf{PSPACE}$. In 1997 Koiran proved that if one assumes the Riemann hypothesis, then 0-1-$\mathbf{NP_C} \subseteq \mathbf{AM}[2]$. Recall that $\mathbf{AM}[2]$ is $\mathbf{BP} \cdot \mathbf{NP}$ so Koiran's result suggests that 0-1-$\mathbf{NP_C}$ may not be much bigger than $\mathbf{NP}$.

Figure unavailable in pdf file.

Figure 14.4: Tableau of Turing Machine configurations

14.3.2 Hilbert's Nullstellensatz

The language $\mathbf{HN_C}$ is defined as the decision version of Hilbert's Nullstellensatz over $\mathbf{C}$. The input consists of m polynomials p_i of degree d over $x_1, \ldots, x_n$. The output is "yes" iff the polynomials have a common root $a_1, \ldots, a_n$. Note that this problem is general enough to include SAT. We illustrate that by the following example:

$$x \vee y \vee z \leftrightarrow (1-x)(1-y)(1-z) = 0.$$

Next we use this fact to prove that the language 0-1-$\mathbf{HN_C}$ (where the polynomials have 0-1 coefficients) is complete for 0-1-$\mathbf{NP_C}$.

THEOREM 14.23 (BSS)
0-1-$\mathbf{HN_C}$ *is complete for* 0-1-$\mathbf{NP_C}$.

PROOF: (Sketch) It is straightforward to verify that 0-1-$\mathbf{HN_C}$ is in 0-1-$\mathbf{NP_C}$. To prove the hardness part we imitate the proof of the Cook-Levin theorem; we create a computation tableau and show that the verification is in 0-1-$\mathbf{HN_C}$.

To that end, consider the usual computation tableau of a Turing Machine over $\mathbf{C}$ and as in the case of the standard Turing Machines express the fact that the tableau is valid by verifying all the 2×3 windows, i.e., it is sufficient to perform local checks (Figure 14.4). Reasoning as in the case of algebraic computation trees (see Lemma 14.16) we can express these local checks with polynomial constraints of bounded degree. The computation states $c \leftarrow q(a,b)/r(a,b)$ are easily handled by setting $p(c) \equiv q(a,b) - cr(a,b)$. For the branch states $p(a,b) \neq 0$ we can use Rabinovitch's trick to convert them to equality checks $q(a,b,z) = 0$. Thus the degree of our constraints depends upon the degree of the polynomials hardwired into the machine. Also, the polynomial constraints use real coefficients (involving real numbers hardwired into the machine). Converting these polynomial constraints to use only 0 and 1 as coefficients requires work. The idea is to show that the real numbers hardwired into the machine have no effect since the input is a binary string. We omit this mathematical argument here. ■

14.3.3 Decidability Questions: Mandelbrot Set

Since the Blum-Shub-Smale model is more powerful than the ordinary Turing Machine, it makes sense to revisit decidability questions. In this section we show that some problems do indeed remain undecidable. We study the decidability of the Mandelbrot set with respect to Turing Machines over $\mathbf{C}$. Roger Penrose had raised this question in his meditation regarding artificial intelligence.

DEFINITION 14.24 (MANDELBROT SET DECISION PROBLEM)
Let $P_C(Z) = Z^2 + C$. Then, the Mandelbrot set is defined as

$$\mathcal{M} = \{C \mid \text{the sequence } P_C(0), P_C(P_C(0)), P_C(P_C(P_C(0)))\ldots \text{ is bounded } \}.$$

Note that the complement of $\mathcal{M}$ is recognizable if we allow inequality constraints. This is because the sequence is unbounded iff some number $P_C^k(0)$ has complex magnitude greater than 2 for some k (exercise!) and this can be detected in finite time. However, detecting that $P_C^k(0)$ is bounded for every k seems harder. Indeed, we have:

THEOREM 14.25
$\mathcal{M}$ is undecidable by a machine over $\mathbf{C}$.

PROOF: (Sketch) The proof uses the topology of the Mandelbrot set. Let $\mathcal{M}$ be any TM over the complex numbers that supposedly decides this set. Consider T steps of the computation of this TM. Reasoning as in Theorem 14.23 and in our theorems about algebraic computation trees, we conclude that the sets of inputs accepted in T steps is a finite union of semialgebraic sets (i.e., sets defined using solutions to a system of polynomial equations). Hence the language accepted by $\mathcal{M}$ is a countable union of semi-algebraic sets, which implies that its Hausdorft dimension is 1. But it is known Mandelbrot set has Hausdorff dimension 2, hence $\mathcal{M}$ cannot decide it. ■

Exercises

§1 Show that if field F is finite then arithmetic circuits have exactly the same power —up to constant factors—as boolean circuits.

§2 Equivalence of circuits of depth d to straight line programs of size $\exp(d)$. (Lecture 19 in Madhu's notes.)

§3 Bauer-Strassen lemma?

§4 If function computed in time T on algebraic TM then it has algebraic computation tree of depth $O(d)$.

§5 Prove that if we give the BSS model (over $\mathbb{R}$) the power to test "$a > 0$?" with arbitrary precision, then all of $\mathbf{P}/poly$ can be decided in polynomial time. (Hint: the machine's "program" can contain a constant number of arbitrary real numbers.)

§6 Shamir's trick?

Chapter notes and history

NEEDS A LOT

General reference on algebraic complexity

P. Brgisser, M. Clausen, and M. A. Shokrollahi, Algebraic complexity theory, Springer-Verlag, 1997.

Best reference on BSS model

Blum Cucker Shub Smale.

Algebraic P and NP from Valiant 81 and Skyum-Valiant'86.

Roger Penrose: emperor's new mind.

Mandelbrot : fractals.