6.966 Algebra and Computation	November, 18 1998
Lecture 20	
Lecturer: Madhu Sudan	*Scribe: Zulfikar Ramzan*

1 Introduction

In this lecture we continue our discussion of algebraic notions of complexity. We define the notion of a Turing Machine over an arbitrary ring, and consider the analogs of various complexity classes in this new model. We discuss some of the relevant open problems in this algebraic setting, and examine what solving them would imply in the traditional setting. We demonstrate a natural complete problem for one of the new complexity classes we consider. We also examine a slightly different way to think about computation in this setting, and using this new way of looking at things, we show that a fundamental and well studied problem from pure mathematics is in fact undecidable.

2 Turing Machines over Arbitrary Rings

As described in the previous lecture, we can define Turing Machines over arbitrary algebraic rings. The traditional Turing Machine model, which examines the ring Z_2, is a special case of this general model. We now define a Turing Machine over an arbitrary ring R. These general algebraic Turing Machines have a two way infinite tape where each cell contains an element of the ring R. We often refer to specific parts of the tape by the notation $Tape(index)$. These general Turing Machines also have a set of states. Some of these states specify a polynomial – and each state falls into one of 5 different categories:

1. Accept States - corresponds to the traditional definition of accept state.
2. Reject States - corresponds to the traditional definition of reject state.
3. Branch States - evaluate the polynomial specified in the definition of the state on the value $Tape(0)$. Branch to another state depending on whether or not the polynomial evaluated to 0.
4. Shift State - shift your entire tape over to the right or left; e.g. for all i: $Tape'(i) = Tape(i+1)$ or $Tape'(i) = Tape(i-1)$.
5. Computing State - perform a computation of the polynomial specified in the definition of the state on value appearing at $Tape(0)$.

Each of these states may contain two different types of information:

1. A finite number of bits describing the name of a state, and other basic information about the state.
2. An finite number of elements from the ring R. These elements actually specify the polynomial whose evaluation on some particular element determines the next action to be taken. For example, if you are in a branch state, depending on what the polynomial evaluates to, you may decide whether or not to take the branch. These polynomials are finitely specified and have a constant arity.

One thing to notice about this definition is that there is conceivably an immense amount of power such a Turing Machine could have if it can record arbitrary ring elements in its states. If we consider the ring of Real numbers, then it is possible to encode an infinitely long string into a state under this model by, for example, having a trancendental number appear as a coefficient in the polynomial specified by a state. Even though this appears to be an overwhelming amount of information, that view is deceptive. In particular, even if you have an infinitely long number encoded in a state, accessing some particular bit of the number, or some particular portion of it, may not be an easy task (because you don't explicitly have the power to determine if a number is greater than or equal to 0). And so it's possible that the task of accessing a particular digit in the number may not be achievable in polynomial time.

We know that a Turing Machine over an arbitrary ring R has a two-way infinite tape which may contain arbitrary elements of the ring R. Since we *do not* place any kinds of restrictions on these elements, it may in fact be possible to place a single element which does not have a finite representation (such as $\sqrt{2}$), on a single cell of the tape, which deceptively appears to give it an enormous amount of power. Having defined these Turing Machines over arbitrary rings, we now shift our focus to the relationships between these arbitrary machines and the classical Turing Machine (which we can think of as working over the ring Z_2). We explore the algebraic analogs of some of the classical complexity theoretic notions.

2.1 Algebraic Turing Machines as Universal Controllers

The first analogous notion between classical complexity theory and algebraic complexity theory that we consider is that of the *Universal Turing Machine.* It's not hard to see that we can consider the notion of a *Universal Turing Machine* even in this algebraic setting. We can design a Universal Turing Machine UTM_R over a ring R which takes as input:

1. The description of a Turing Machine M_R over the ring R.
2. An input x to this Turing Machine M_R.

The Universal Turing Machine should then simulate M_R on input x and output the same thing as M_R would have. The design of such a Universal Turing Machine is not too difficult. We now move on to consider the algebraic analog of decision problems.

2.2 Decision Problems

We'll be interested in decision problems which are a subset of instances of Algebraic Turing Machines. In particular consider an infinite family of functions $\{f_n\}_{n\geq 0}$ where each $f_n : R^n \rightarrow R$ is a function induced by an Algebraic Turing Machine that operates over the ring R. We can think of these functions as decision problems where we can think of x such that $f(x) = 1$ as a *yes* instance, and everything else as a *no* instance.

2.3 Comparison to Traditional Complexity Theoretic Models

We now compare these algebraic models and their variants with the traditional $\{0, 1\}$ model, and we show that some surprising results can occur if we make minor, yet seemingly reasonable, modifications to these models. To start with let's define the notion of a polynomial time Turing Machine over a ring R to be one that runs in time polynomial in the input size – in this case when we say input size, we mean the number of cells used in the input tape. Now, we show that adding what appears to be a tiny, yet reasonable, amount of functionality to polynomial time Algebraic Turing Machines enables them to solve undecidable problems in polynomial time!

2.3.1 Solving Undecidable Problems!

Suppose we allow these Turing Machines the power to tell whether or not a given number is greater than or equal to 0. Call such Turing Machines: *Turing Machines with Inequality.* It turns out that even this tiny and reasonable amount of extra functionality enables these Turing Machines to solve undecidable problems in a polynomial amount of time. Here is the idea. We know that the class $P/poly$[1] contains undecidable problems.

[1] recall that $P/poly$ is the class of languages C such that L is in C if there exists a polynomial time Turing Machine M and a sequence of strings $\{s_i\}_{i\in N}$ such that $|s_i| = i^{O(1)}$ and $x \in L$ if and only if $M(x, s_{|x|})$ accepts.

Now, if we use algebraic Turing Machines over the Reals, then it's possible to encode a polynomial length advice string within a single state, as a single element of the ring. If we allow the computation of ≥ 0 then accessing the $i-th$ digit of this advice string becomes easy (if we don't allow the operation $\geq$ this task may be very difficult). In light of this, it turns out that the Algebraic Turing Machine defined over the Reals can solve every problem in $P/poly$, thus they can solve undecidable problems.

The Turing Machine over the Reals with Inequality (≥ 0) is not a terribly well behaved model because it can solve classically undecidable problems in polynomial time. This model may not always be interesting depending on how you look at things. One thing we can do to make this model interesting is to consider specific problems and specific problem instances and see how well these Turing Machines solve these problems.

2.3.2 Considering Specific Problems

If we fix certain problems of interest, we can consider various questions about how well a Turing Machine over the Reals with Inequality might handle these problems. For example, some interesting questions we can consider in this model are:

1. Is the Mandelbrodt Set Decidable by a Turing Machine over the Reals with Inequality? (We will formally define and answer this question later in the lecture)

2. Does Linear Programming have a strongly polynomial time algorithm in this same setting?

So we can fix problems of interest and ask how well these problems can be solved by such Turing Machines. In the case of Linear Programming, we might define an instance of Linear Programming as taking input $< A, b >$ and trying to find a vector x such that each element of x is greater than 0 and $Ax \leq b$.

Defining Linear Programming in this manner begs the question of what constitutes a good encoding. For example, it's possible to encode an instance of Linear Programming in the space of a few ring elements (since our ring elements may be infinitely long, and can encode a lot of information). And in general, for any problem, it might be possible to encode an instance of the problem within a single real number. The best way to handle this question is to recall that we undergo a similar dillema in the traditional 0/1 case. In particular, under some encodings, even very hard problems become trivial to solve. So, all we can do is make a qualitative assessment that a particular encoding of a problem that we're considering is "reasonable" and that it makes "the most sense." Having studied the algebraic analogs to some fundamental classical notions, we are in a position to examine the corresponding complexity classes that occur.

2.3.3 Defining Analogs to Traditional Complexity Classes

We now define some traditional complexity classes in this new Algebraic setting, and consider some of the interesting problems that arise. First we consider the analog of the class NP. Traditionally, one can define NP to be the set of projections of languages on two inputs that are in P. We can carry this analogy over to the setting of Algebraic Turing Machines. Consider a language L that consists of pairs (x, y) where $x \in R^n$ and $y \in R^{poly(n)}$ for some polynomial $poly(n)$, and where L is described by some polynomial time $R-TM$ called M_R. (We use $R-TM$ to denote a Turing Machine which is defined over the ring R.) We now define the projection of L

Definition 1 *Let L be a language. Then the Projection of L, denoted $\Pi(L)$ is defined as:*

$$\Pi(L) \equiv \{x \mid \exists\ y \text{ such that } (x, y) \in L\} \tag{1}$$

We now formally define the class NP_R:

Definition 2 $NP_R \equiv \{\Pi(L) \mid L \in P_R\}$. *Where P_R is the set of languages decidable in polynomial time by some $R-TM$.*

Now that we've defined NP_R, we can ask the natural question: $NP_R =? P_R$? Just like in the traditional scenario, this is an open problem, and later we give strong evidence for why it is believed that $NP_R \neq P_R$.

We now describe some other problems that are open – first we consider the notion of restrictions of traditional complexity classes to binary settings:

Definition 3 *Let C be any complexity class; i.e. C is a collection of languages $\{L\}$ where $L \in R^*$ (where $R^* \equiv \bigcup_{n\geq 0} R^n$). Then, we define $0/1 - L$ as:*

$$0/1 - L \equiv L \bigcap \{0,1\}^* \tag{2}$$

And we define $0/1 - C$ as:

$$0/1 - C \equiv \{0/1 - L \mid L \in C\} \tag{3}$$

This notion gives rise to another open question: does $0/1 - P_R =? P$?. It's possible that $0/1 - P_R$ is stronger than than P because, for example, you can compute 2^{2^n} in n steps with an $R - TM$ (e.g. in successive steps you can compute: $2, 2^2, 2^4, 2^8, \ldots$ via repeated squaring). Can such a computation help you? It's not so obvious because even if you can perform such a computation, it's hard to extract any particular piece, such as the $i - th$ bit of the computation efficiently if you aren't given the power to perform inequality checks. It turns out that we do know one thing about $0/1 - P_R$ if R is the Ring of Complex Numbers then:

Theorem 4 *Let $\mathcal{C}$ denote the ring of complex numbers. Then, $0/1 - P_{\mathcal{C}} \subseteq BPP$.*

This gives evidence that $0/1 - P_{\mathcal{C}}$ may not be bigger than P. We now look specifically at the class of Turing Machines defined over the Complex Numbers.

2.4 Turing Machines over the Complex Numbers

We consider the class $\mathcal{C}$ of Complex Numbers, and consider various complexity classes associated with Turing Machines that are defined over the complex numbers. Let's start by considering $0/1 - NP_{\mathcal{C}}$. This is an interesting class because you can have arbitrarily high precision computation, and arbitrarily high precision witnesses! In particular, we show that Hilbert's Nullstellensatz when defined appropriately over 0/1 is $NP_{\mathcal{C}}$-complete. First let's define an appopriate decision version of Hilbert's Nullstellensatz when defined over an arbitrary field K.

0/1-Hilbert's Nullstellensatz (HN)
Input: polynomials $f_1, \ldots f_n \subseteq K[x_1, \ldots x_n]$. Where each polynomial is defined on n variables $x_1, \ldots, x_n$ and is defined over the field K. Moreover, these polynomials have degree bound d.

Question: Are there inputs $a_1, \ldots, a_n$ such that $f_i(a_1, \ldots, a_n) = 0$?

Given this formulation, one can talk about $0/1 - HN_{\mathcal{C}}$. We will later show that $0/1 - HN_{\mathcal{C}}$ is $0/1 - NP_{\mathcal{C}}$ complete. Before doing so, let us examine some of the ramifications.

Theorem 5 $0/1 - NP_{\mathcal{C}} \subseteq PSPACE$

Proof Will follow directly from the proof that $0/1 - HN_{\mathcal{C}}$ is $0/1 - NP_{\mathcal{C}}$-complete. ■

Theorem 6 *If $0/1 - P_{\mathcal{C}} = 0/1 - NP_{\mathcal{C}}$ then $NP \subseteq BPP$.*

Proof If $0/1 - P_{\mathcal{C}} = 0/1 - NP_{\mathcal{C}}$ then $0/1 - HN_{\mathcal{C}} \in P_{\mathcal{C}}$. Now, we can think of $3 - SAT$ as a special case of $0/1 - HN_{\mathcal{C}}$. So we have that $3SAT \in 0/1 - NP_{\mathcal{C}} = 0/1 - P_{\mathcal{C}} \subseteq BPP$. ■

Theorem 7 *If $0/1 - P_{\mathcal{C}} \neq 0/1 - NP_{\mathcal{C}}$ then $P \neq PSPACE$.*

Proof If $0/1 - P_{\mathcal{C}} \neq 0/1 - NP_{\mathcal{C}}$ then $0/1 - HN_{\mathcal{C}} \notin 0/1 - P_{\mathcal{C}}$. Therefore, $P \neq PSPACE$ since $0/1 - HN_{\mathcal{C}} \in PSPACE$. ■

It is open whether $0/1 - P_{\mathcal{C}} \neq 0/1 - NP_{\mathcal{C}}$ would imply that $P \neq NP$. There is a recent result that $0/1 - HN_{\mathcal{C}} \subseteq PH$ under the extended Riemann Hypothesis. [2]

We are now ready to state and prove our main theorem:

Theorem 8 *$0/1 - HN_{\mathcal{C}}$ is $0/1 - NP_{\mathcal{C}}$-complete.*

Proof It is easy to see that $0/1 - HN_{\mathcal{C}} \in 0/1 - NP_{\mathcal{C}}$ because we can make our witness the vector $a_1, \ldots, a_n$ that makes all the polynomials evaluate to 0 (e.g. $f_i(a_1, \ldots, a_n) = 0$. Now, to show that $0/1 - HN_{\mathcal{C}}$ is $0/1 - NP_{\mathcal{C}}$-hard, we show that we can create a Cook-like tableau and do verification in the usual way.

Consider the computation of a $\mathcal{C} - \mathcal{TM}$ that decides a language in $0/1 - NP_{\mathcal{C}}$. For each time step $t = 0, t = 1, \ldots, t = T$ we can consider the usual configurations on the Turing Machine tape, and ask ourselves the usual question of whether or not this computation is valid. More specifically to this case, we need to ask whether or not this computation can be expressed in terms of low degree polynomials. What kinds of information are contained in the configuration of this Turing Machine at a given time step? First there are basic bits of information like what the position of the head is, what state you're in, etc,. Next, there may be arbitrary elements of the ring R in the configuration. The first case is easy to deal with, while the latter requires more work. The goal is to do local checks over the iterations to see whether or not a particular configuration correctly follows from the previous configuration. We need to check two cases: 1) the current state is a computation state and 2) the current state is a branch state. All other cases can be handled in the standard way. If the current state is a computation state, then all the checks are easily expressible by some condition of the form $poly = 0$; i.e. does a given polynomial evaluate to 0? The only difficult case is if we're in a branch state. In this case, we may have some condition of the form: $p(a_1, a_2, a_3) =? 0?$. Where if the condition is true, then we go to state i and if the condition is false, then we should go to state j. So, we want to be able express this kind of computation. We can express it in terms of two polynomials $q(x)$ and $r(x)$ where:

$$q(x) = 0 \Longrightarrow p(a_1, a_2, a_3) = 0$$

$$r(x) = 0 \Longrightarrow p(a_1, a_2, a_3) \neq 0$$

And, using the usual manipulations, we can come up with a polynomial $P(a_1, a_2, a_3, x)$ such that if $P(a_1, a_2, a_3, x) \neq 0$ then $r(x) = 0$ and if $P(a_1, a_2, a_3, x) = 0$ then $q(x) = 0$. So, we have now defined the following scenario:
1. We have polynomials $p_1, \ldots, p_l \in R[x_1, \ldots, x_n]$
2. And we have polynomials $q_1, \ldots, q_k \in R[x_1, \ldots, x_n]$
And we need to answer the following question:

Does there exist $a_1, \ldots, a_n \in R$ such that $p_i(a_1, \ldots, a_n) = 0$ and $q_j(a_1, \ldots, a_n) \neq 0$

Now, this is close to the Nullstellensatz – and to complete the reduction, we just have to show how to convert all the inequalities to equalities. We can use Rabinowitsch's Trick! So, given polynomials $p_1, \ldots, p_l$ and polynomials $q_1, \ldots, q_k$, we can define polynomials $p_1, \ldots, p_l$ and r where:

$$r(x_1, \ldots, x_n, y) \equiv (1 - y \cdot \Pi_{i=1}^{k} q_j(x_1, \ldots, x_n))$$

Now we can make all equality checks! And so we've transformed any input instance into one of the 0/1 Hilbert Nullstellensatz. Therefore, $0/1 - HN_{\mathcal{C}}$ is $0/1 - NP_{\mathcal{C}}$-complete. ■

[2] The extended Riemann Hypothesis conjectures that the first quadratic non-residue mod p of a number is always less than $2(\ln p)^2$. This implies something along the lines of: every prime number has a generator from a small initial set.

One thing to note about this proof is that Rabinowitsch's trick requires that you work in a field (because you need inverses which may not exist if you work over a ring). Consequently, this result only holds over arbitrary fields but does not necessarily hold over arbitrary rings. Having examined one way to view the computation of these Algebraic Turing Machines, we now shift to a slightly different point of view: the computation tree.

2.5 Other Ways to View Computation

We consider other ways to view our new model of Algebraic computation. The fundamental question we are interested in is: given fixed inputs $x_1, \ldots, x_m$ to an $R-TM$ M_R, how can you reason about this computation? In particular, we will consider the *computation trees* of $R-TMs$ and use them to better understand computation in this Algebraic setting. Observe that we can view most of the computation in terms of some sufficiently high degree polynomial. Suppose that the first k steps of the computation in an $R-TM$ don't involve branches. Then, you can find polynomials $P_1, \ldots, P_k$ (each of which represents one step of the computation) where:

$$y_1 = P_1(x_1, \ldots, x_n)$$
$$y_2 = P_2(x_1, \ldots, x_n, y_1)$$
$$y_3 = P_3(x_1, \ldots, x_n, y_2)$$
$$\vdots$$
$$y_k = P_k(x_1, \ldots, x_n, y_{k-1})$$

And with a little work you can find polynomials $P_1', \ldots, P_k'$ such that $y_1 = P_1'(x_1, \ldots, x_n), \ldots, y_k = P_k'(x_1, \ldots x_k)$ (though the P_i' may have higher degree than the corresponding P_i). So, in particular, all non branch steps can be compressed and represented in terms of a single polynomial. This polynomial will have degree at most 2^{2^k} where k is the number of steps in your non-branching block of code. The bound 2^{2^k} is derived from the fact that you start with degree 2 polynomials, and at each step your polynomials can at most double in degree.

Given this formulation, if we add on branching states, then we can view all of the computation in terms of a "compressed" tree. Where each node contains a polynomial that represents either one of the above mentioned compressed non-branching polynomials, or a branching polynomial. Each branching node will then have pointers to two nodes (depending on whether or not the branch is taken). This alternative way of looking at computation enables us to attack a well known problem from pure mathematics: the decidability of the Mandelbrodt Set.

2.6 Is the Mandelbrodt Set Decidable?

We now consider the decidability of the Mandelbrot set under Algebraic Turing Machines over the complex numbers. The Mandelbrodt Decidability problem is formally defined as follows:
Input: A single complex number c.
Question: Consider the function $P_c : z \mapsto z^2 + c$. Is the following sequence bounded:

$$P_c(0), P_c(P_c(0)), P_c(P_c(P_c(0))), \ldots ?$$

(i.e. is c such that the iteration remains bounded?)
Traditional mathematics has indicated to us that this problem is very hard to solve – it is difficult to get a precise form on the value c. Now we have the tools to reason about this in a natural fashion – this is a computational problem that can be addressed using Turing Machines over the Reals. We can phrase this problem in the following equivalent fashion. If we define $Z_i \equiv P_c^i(0)$ then we can ask if there is a value k such that $|Z_k| > 2$. This condition is equivalent to asking whether or not the sequence is bounded because

if $|Z_k| > 2$ then it's not difficult to work out that the sequence will diverge, and conversely if the sequence is unbounded, then it trivially follows that at some point $|Z_k| > 2$.

Now, if the sequence in unbounded – then k such that $|Z_k| > 2$ can be found in finite time. However, if the condition is not true, then this becomes more difficult. In fact, it turns out that this problem is *undecidable*. We outline the proof below:

Theorem 9 *The language* $\{c \mid \text{ the sequence } Z_1, Z_2, \ldots \text{ converges}\}$ *is undecidable.*

Proof (Sketch) Acceptance conditions by a Real Turing Machine at time T can be given by a collection of polynomial inequalities:

$$f_1(x_1, \ldots, x_n) \geq 0$$
$$f_2(x_1, \ldots, x_n) \geq 0$$
$$f_3(x_1, \ldots, x_n) \geq 0$$
$$\vdots$$
$$f_k(x_1, \ldots, x_n) \geq 0$$

And this defines a semi-algebraic set. Now:

$$\bigcup_{t \geq 0} \text{acceptance at time } t$$

is a countable union of algebraic sets. Moreover, it is known that the boundary of a semi-algebraic set is a variety in R^2 and therefore it has Haussdorf Dimension 1. Since countable unions of semi-algebraic sets preserve Haussdorf dimension the overall Haussdorf dimension is 1. Now, let's shift to what is known about Mandelbrodt sets. It turns out that the Mandelbrodt set is connected, and has Haussdorf dimension 2 – hence by the above argument, it cannot be decidable. This proof is remarkable because we translated known topological constructs into complexity theoretic constructs. ■