Math 278 - Geometry and Algebra of Computational Complexity

Taught by David Donghoon Hyeon
Notes by Dongryul Kim

Fall 2018

¡+instructor+¿ ¡+meetingtimes+¿ ¡+textbook+¿ ¡+enrolled+¿ ¡+grading+¿ ¡+courseassistants+¿

Contents

1 September 5, 2018 — 2
- 1.1 Turing machines . . . 2

2 September 10, 2018 — 4
- 2.1 Non-deterministic Turing machines . . . 4
- 2.2 Encoding Turing machines . . . 5

3 September 12, 2018 — 7
- 3.1 Uncomputable functions . . . 7

4 September 17, 2018 — 8
- 4.1 Satisfiablity . . . 8
- 4.2 Hilbert's Nullstellensatz . . . 9
- 4.3 Hilbert's tenth problem . . . 10

5 September 19, 2018 — 11
- 5.1 Decidable and semi-decidable sets . . . 11
- 5.2 Register machines . . . 12

6 September 24, 2018 — 14
- 6.1 Register equations and their Diophantization . . . 14

7 September 26, 2018 — 16
- 7.1 Diophantization of the exponential relation . . . 16
- 7.2 Finishing Hilbert's tenth problem . . . 17

1 September 5, 2018

There are going to be biweekly homeworks, and a final writing project. The goal of the course is to introduce you to the various aspects of computational complexity theory. There will be four parts:

1. Turing machines, deterministic and non-deterministic, probabilistic algorithms, reduction, NP-completeness
2. Undecidable problems, Hilbert's 10th problem of solving diophantine equations
3. Computer models, continuous time systems, Blum–Smale–Shub model, quantum computers
4. Geometric complexity theory, algebro-geometric and representation theoretic approach to P≠NP

We may consider the determinant as a point in $\mathbb{P}(\mathrm{Sym}^n(\mathbb{C}^{n^2}))$. There is this conjecture that there is no constant $c \geq 1$ such that for all large m,

$$\mathrm{GL}_{m^{2c}}[\ell^{m^c - m}\,\mathrm{perm}_m] \notin \overline{\mathrm{GL}_{m^{2c}}[\det_{m^2}]}.$$

This implies P≠NP.

When you do any kind of programming at home, you use discrete time and discrete space. At the end, it really looks like

$$x_{k+1} = f(x_k).$$

On the other hand, the continuous time and space analogue will be a differential equation

$$y' = f(y).$$

Differential analyzers and continuous neural networks are like this. On the other hand, states in quantum computers lie in Hilbert spaces, and so they have continuous space but discrete time.

1.1 Turing machines

This is going to be boring. Let Σ be a finite set of alphabets, for instance, $\Sigma = \{0, 1\}$ for modern computers. Σ^* is the set of all words on Σ.

Definition 1.1. A **language** over Σ is a subset of Σ^*. A **decision problem** encoded on Σ is a partition

$$\Sigma^* = (\text{yes}) \amalg (\text{no}) \amalg (\text{non}).$$

(You get a yes or a no or an error.) The language associated to a decision problem Π is the "yes" part, and is denoted by L_Π.

Definition 1.2. A **deterministic Turing machine** has a read-write had, a bi-infinite tape, and a DTM program consisting of

- Σ a finite set of tape symbols, with $b \in \Sigma$ a blank symbol, and $\gamma \subseteq \Sigma$ a set of input symbols with $b \notin \gamma$,
- a finite set Q of states with distinguished q_0, q_Y, q_N of start, yes, no states,
- a transition function

$$\delta : (Q \setminus \{q_Y, q_N\}) \times \Sigma \to Q \times \Sigma \times \{\pm 1\}.$$

You should think of there being an infinite tape and a state-controller pointing to a certain point on the tape. The state-controller reads the tape symbol at that point, and plugs its own state and the tape symbol to δ. The output will be the new state of the state-controller, the symbol that will be written, and where the read-write head will move next. The program ends when either q_Y or q_N is hit.

On some inputs, a deterministic Turing machine may never halt. In fact, there is no "algorithm" that can determine whether a given deterministic Turing machine halts on a certain input. We will prove this shortly.

Example 1.3. Consider the following Turing machine. Find what this does.

$q \setminus \sigma$	0	1	b
0	$0, 0, 1$	$0, 1, 1$	$1, b, -1$
1	$2, b, -1$	$3, b, -1$	$N, b, -1$
2	$Y, b, -1$	$N, b, -1$	$N, b, -1$
3	$N, b, -1$	$N, b, -1$	$N, b, -1$

Definition 1.4. Let M be a deterministic Turing machine. The language recognized by M is

$$L_M = \{x \in \gamma^* : M \text{ accepts } x\}.$$

So M solves the decision problem Π if $L_M = \Pi$.

Definition 1.5. The **time complexity** of M is the function

$$T_M(n) = \max_{|x|=n} (m : M \text{ halts on } x \text{ in } m \text{ steps}),$$

where a step is a movement of the head.

2 September 10, 2018

Today we will talk about non-deterministic Turing machines.

2.1 Non-deterministic Turing machines

I will give two definitions, which are going to be equivalent. Recall that a deterministic Turing machine is just a infinite tape with a read-write head. The program really is the transition function $\delta : Q \setminus \{q_Y, q_N\} \times \Gamma \to Q \times \Gamma \times \{\pm 1\}$. In a **non-deterministic Turing machine**, the picture is the same, but there are two transition functions δ_0 and δ_1. At each computational step, the machine makes an arbitrary choice between δ_0 and δ_1.

Definition 2.1. A **computation path** is the sequence of choices the machine makes. For instance, it looks like

$$\delta_0\delta_1\delta_0\delta_0\delta_1\delta_1 \cdots \text{ or } 010011 \cdots .$$

The length of the computation path is going to be the length of the computation.

Definition 2.2. M is said to run in time $T(n)$ if for every input x and every computation path, the machine halts within $T(|x|)$ steps. We say that M is a **polynomial time** non-deterministic Turing machine if it runs in some polynomial time.

We say that M accepts x if there exists a computation path that halts with q_Y. Then we define the language accepted by M as

$$L_M = \{x \in \Sigma^* : M \text{ accepts } x\} \subseteq \Sigma^*.$$

Then we define

$$\mathcal{NP} = \{L \subseteq \Sigma^* : \text{exists a polynomial nDTM } M \text{ with } L_M = L\}.$$

It is clear that $\mathcal{P} = \mathcal{NP}$, because a DTM is always a nDTM. ($\mathcal{P}$ is the same thing with DTM instead of nDTM.) Intuitively, $\mathcal{NP}$ means that you can check an answer (computational path) in polynomial time.

Let me give an alternative definition of an nDTM. We now consider a two-tape machine, and we consider a transition function

$$\delta : Q \times \Gamma \times \Gamma \to Q \times \Gamma \times \Gamma \times \{\pm 1\} \times \{0, 1\}.$$

It also has a "guessing module". On an input x on the first tape, the guessing module writes an arbitrary guess y on second tape, of length bounded in polynomial by the length of x. Then the machine proceeds with the computation deterministically.

Definition 2.3. We say that M runs in time $T(n)$ if on an input x and for any guess, M halts in $T(|x|)$ steps.

Using this, we can again define $\mathcal{NP}$ so that L is in $\mathcal{NP}$ if there exists a language R (recognizable by a polynomial DTM) and a polynomial q such that

$$L = \{x : \exists y, |y| \leq q(|x|), (x, y) \in R\}.$$

In this case, we say that y is a "witness" or a "certificate" for x.

Theorem 2.4. *The two definitions are equivalent.*

Proof. Let L be $\mathcal{NP}$ according to the first definition. Then you can use the computation path as the guess. In particular, we can do something like

$$\delta(q, \sigma_1, \sigma_2) = (\sigma_2 \delta_1(\sigma_1, q) + (1 - \sigma_2)\delta_0(\sigma_1, q), 1).$$

The other direction does it similarly. $\square$

You can also define stuff like k-tape machines, but if you thing hard enough, you will see that there is no difference.

Definition 2.5. We say that a problem Π is **reduced** to Π' if there is a (polynomially) computable funciton

$$f : \Sigma^* \to \Sigma^*$$

such that $x \in L(\Pi)$ if and only if $f(x) \in L(\Pi')$.

What do we mean by a computable function? The easiest way to define it is by using a k-tape machine. This k-tape machine M has a dedicated input tape and an output tape. We say that M computes f if on input x, the content of the output tape is equal to $f(x)$ when the machine halts.

Definition 2.6. A problem or language is said to be **NP-hard** if any NP language can be polynomially reduced to it. It is said to be **NP-complete** if it itself is in NP.

If you search on Wikipedia, you can find hundreds of examples of NP-complete problems, mostly in discrete mathematics.

2.2 Encoding Turing machines

Now we want to encode a Turing machine, i.e., construct a map

$$\epsilon : \{0, 1\}^* \to \{\text{Turing machines}\}.$$

We are going to make a Turing machine on $\{0, 1, -\}$ and $Q = \{0, 1, 2, \ldots, l\}$. We encode ℓ and the transition function from values $\delta(\sigma, q)$ as a binary word. If any binary string does not come from this procedure, map it to some trivial Turing machine. This defines ϵ.

Definition 2.7. There exists a DTM $\mathcal{U}$ such that for every (x, α),

$$\mathcal{U}(x, \alpha) = M_\alpha(x).$$

This is called the **universal Turing machine**. If M_α halts on input x within T steps, then $\mathcal{U}$ halts in (x, α) within $CT \log T$ steps.

Our personal computers are all like this. If you write a program, you can run it. You can see at a high level how this will work. I was told that it is very involved to actually construct this machine.

3 September 12, 2018

We will only have 30 minutes of lecture because there is the Ahlfors lectures.

3.1 Uncomputable functions

If you want to show that uncomputable functions exists, this is easy because there are countably many Turing machine, and uncountably many languages. So we want a construction of a function that is not computable by any DTM.

Example 3.1. Recall that we had this encoding of a DTM given by

$$\epsilon : \Sigma^* \to \{\text{DTMs}\}; \quad \alpha \mapsto M_\alpha.$$

Now define

$$f(\alpha) = \begin{cases} 0 & M_\alpha \text{ accepts } \alpha, \\ 1 & \text{else.} \end{cases}$$

Then we claim that f is not computable. Suppose that $M = M_{\alpha^*}$ computes f. Then

$$M_{\alpha^*}(\alpha^*) = 1 \quad \Leftrightarrow \quad f(\alpha^*) = 1 \quad \Leftrightarrow \quad M_{\alpha^*} \text{ does not accept } \alpha^*.$$

This is contradictory.

Example 3.2. Here is another example. Consider the problem of taking (α, x) and outputing whether M_α halts on input α. Suppose M_ξ solves the Halting problem HALT. We are then going to build a solution to the previous function by using the universal Turing machine. You first plug in (α, α) to M_ξ, and if it says no, just output 1. If it says yes, run $\mathcal{U}$ with α and α, and output the answer. This shows that the halting problem is undecidable.

Example 3.3. Let us look at the Bounded Halting Problem for nDTMs, denoted BHPN. First note that nDTMs can be encoded,

$$\epsilon : \Sigma^* \to \{\text{nDTMs}\},$$

and also that there is an efficient universal nDTMs. Now the input is (α, x, t), and the problem is,

> Does M_α halt on x on t steps?

This problem is $\mathcal{NP}$ because we can use the universal machine. On the other hand, it is $\mathcal{NP}$-hard as well. To see this, let $L \in \mathcal{NP}$ and let M be the nDTM that recognizes L. Then we can define

$$f : \Sigma^* \to \Sigma^*; \quad x \mapsto (\alpha, x, T(|x|)).$$

This reduces L to the Bounded Halting Problem. This shows that BHPN is $\mathcal{NP}$-complete.

4 September 17, 2018

Last time we constructed an uncomputable function. The point was to give an explicit construction. This was

$$f(\alpha) = \begin{cases} 0 & M_\alpha \text{ accepts } \alpha \\ 1 & \text{else.} \end{cases}$$

Then we showed that HALT is uncomputable by reducing it to this function. Then, we showed that BHNP is $\mathcal{NP}$-complete. This problem was defined by

$$\{(\alpha, x, t) : M_\alpha \text{ accepts } x \text{ within } x \text{ steps}\}.$$

Now we want a natural problem that is $\mathcal{NP}$-complete.

4.1 Satisfiablity

Let Γ be a finite set of variables. Then a **literal** is a variable x or a negation of a variable $\neg x$. A **clause** is a finite set of literals. A **truth assignment** is a map $\xi : \Gamma \amalg \neg\Gamma \to \{0, 1\}$ such that $\xi(\neg x) = \tau\xi(x)$. An instance of the problem SAT is a finite set I of clauses, and the problem is,

> Does there exist a truth function ξ satisfying all $C \in I$, where ξ satisfies $C = \{U_1, \ldots, U_l\}$ means that $\xi(U_i) = 1$ for some i?

Using the logical "and" $\wedge$ and "or" $\vee$, we can write it as finding a solution to

$$\bigwedge_{C_i \in I} (U_{i1} \vee U_{i2} \vee \cdots \vee U_{ij_i}).$$

Theorem 4.1 (Cook, 1971)**.** *The problem* SAT *is* $\mathcal{NP}$*-complete.*

Proof. It is easy to show that it is $\mathcal{NP}$, because we can set the guess as the truth function. Now let us show that it is $\mathcal{NP}$-hard. Suppose $L \in \mathcal{NP}$ is recognized by a nDTM M. Assume that the tape symbols are $\{0, 1, -1 = \text{blank}\}$, and states $\{0 = q_0, 1 = q_Y, 2 = q_N, \ldots, l\}$. Let the input be x, with $n = |x|$, and assume the running time is $p(n)$.

Now what we are going to do is the write down everything in the computation and turn it into a single formula. Define the logic variables

$$\begin{aligned} \sigma_{t,i,j} &= \text{at time } t, \text{ the tape content in the } i\text{th square is } j, \\ q_{t,s} &= \text{at time } t, \text{ state is } s, \\ h_{t,i} &= \text{at time } t, \text{ head is at tape square } i. \end{aligned}$$

Here, the number of variables is at most constant times $p(n)^2$. Next we can write down all the relations between the variables that we need for it to accept the input. These are

- $q_{0,0}$,

- $q_{p(n),1}$,
- σ_{0,i,x_i} for $1 \leq i \leq n$,
- $\sigma_{0,i,-1}$ for $i \leq -q(n)$ and $i \geq n+1$, (the squares between $-q(n)$ and 0 is used to store the guess)
- $\bigvee_i h_{t,i}$,
- $\neg h_{t,i} \vee \neg h_{t,j}$ for $i \neq j$,
- $\bigvee_j \sigma_{t,i,j}$,
- $\neg\sigma_{t,i,j} \vee \neg\sigma_{t,i,j'}$ for all $j \neq j'$.
- equations encoding the transition functions like

$$\begin{aligned} &\neg\sigma_{t,i,j} \vee \neg h_{t,i} \vee \neg q_{t,s} \vee \sigma_{t+1,i,j'}, \\ &\neg\sigma_{t,i,j} \vee \neg h_{t,i} \vee \neg q_{t,s} \vee \sigma_{t+1,i,j'}, \\ &\neg\sigma_{t,i,j} \vee \neg h_{t,i} \vee \neg q_{t,s} \vee \sigma_{t+1,i,j'}, \end{aligned}$$

  and equations stating that nothing else changes.

You can count the number of variables, and then you are going to see that the number of equations is polynomial in n. $\square$

4.2 Hilbert's Nullstellensatz

Consider an algebraically closed $k = \bar{k}$. Here is a weak version of Hilbert's Nullstellensatz.

Theorem 4.2. *If $f_1, \ldots, f_m \in k[x_1, \ldots, k_n]$, then*

$$f_1 = f_2 = \cdots = f_m = 0$$

has no solution if and only if there exist $g_i \in k[x]$ such that $\sum f_i g_i \equiv 1$.

Now consider the problem HN_k, which have instances $f_1, \ldots, f_m \in k[x]$, and ask,

Does $f_1 = \cdots = f_m = 0$ have a common solution?

If we use Hilbert's Nullstellensatz, we get a linear algebra problem by writing down the coefficients. If we write $f_i = \sum_\alpha a_{i\alpha} x^\alpha$ and $g_i = \sum b_{i\beta} x^\beta$, then we are solving

$$\sum_{\alpha+\beta=\gamma} a_{i\alpha} b_{i\beta} = \begin{cases} 1 & \gamma = 0 \\ 0 & \gamma \neq 0. \end{cases}$$

But what is the size of the system?

Theorem 4.3 (Browawell, Kollar)**.** *We can further impose* $\deg(g_i) \leq O(d^n)$, *where* $d = \max\{3, \deg(f_i)\}$.

In fact, we are going to show that HN is $\mathcal{NP}$-hard, and $\mathcal{NP}$-complete over a finite field. This is an important basis for security analysis in cryptography.

Theorem 4.4. HN *is $\mathcal{NP}$ hard.*

Proof. We will reduce SAT to HN. An instance looks like

$$\bigwedge(u_{i1} \vee \cdots \vee u_{is_i}),$$

and so we consider the system of polynomial equations

$$f_C = \prod f_i$$

for each $C \in I$. □

4.3 Hilbert's tenth problem

This is trying to solve Diophantine equations. A Diophantine equation is,

$$P(x_1, x_2, \ldots, x_n) = 0$$

where $P \in \mathbb{Z}[x_1, \ldots, x_n]$. Then Hilbert's question was to find an algorithm for determining whether a given $P = 0$ has a solution in rational integers.

Definition 4.5. A set $S \subseteq \mathbb{N}^n$ is said to be **Diophantine** if there exists a (integer coefficient) polynomial P such that

$$S = \{a \in \mathbb{N}^n : \text{there exists } \underline{x} \in \mathbb{N}^m \text{ such that } P(a, \underline{x}) = 0\}.$$

For instance, $\{(a, b) : a \geq b\}$ is $\{(a, b) : \exists x, a = b + x\}$, and so Diophantine. The set of composites is

$$\{a : \exists x, y, (a = (x + 2)(y + 2))\}.$$

The set of primes also happens to be prime, and this is a consequence of the Hilbert's tenth problem.

5 September 19, 2018

To show the $\mathcal{NP}$-completeness of SAT, we assigned a bunch of variables to decide the "configuration". Then we encoded what it means to compute, as relations between these variables. This gave a polynomial reduction of any $\mathcal{NP}$ problem to SAT.

5.1 Decidable and semi-decidable sets

Then we defined Diophantine sets as sets S that can be expressed as

$$S = \{a \in \mathbb{N}^m : \text{there exists } x \in \mathbb{N}^n \text{ such that } P(a,x) = 0\}$$

for some polynomial $P(a,x)$. We saw the examples $\{(a,b) : a \geq b\}$ and $\{\text{composites}\}$. A more interesting example is $\{(x,y,n) : x^n + y^n = z^n\}$. In fact, we are going to see that all sets that are algorithmically determinable are Diophantine.

We say that Hilbert's 10th problem is decidable (resp. undecidable) over R if there is (resp. is not) an algorithm for deciding whether a given Diophantine equation has a solution in R. Also, let us denote Hilbert's 10th problem by $\mathsf{H10}$. Hilbert's hope was that $\mathsf{H10}$ is decidable over $\mathbb{Z}$. Then it is also decidable over $\mathbb{Q}$.

Theorem 5.1 (Davis–Putnam–Robinson–Matiyasevich)**.** *The problem* $\mathsf{H10}$ *is undeciable over* $\mathbb{Z}$*.*

Definition 5.2. A set S is **decidable** if there is a deterministic Turing machine that computes χ_S.

For example, $L(\mathsf{HALT})$ is not a decidable set. But we can extend this a bit further.

Definition 5.3. A set S is **semi-decidable** if it is the halting set of a deterministic Turing machine.

Because $L(\mathsf{HALT})$ is the halting set of the universal DTM, it is semi-decidable. This is a really important ingredient in the proof of Hilbert's 10th problem.

Definition 5.4. We say that S is **recursively enumerable** if there exists a deterministic Turing machine M that outputs $x_1\#x_2\#x_3\#\cdots$ where S is precisely the set $S = \{x_1, x_2, \ldots\}$. In other words, S is the range of a computable function.

Proposition 5.5 (homework)**.** *Recursive enumerability is equivalent to semi-decidability.*

Theorem 5.6 (Davis–Putnam–Robinson–Matiyasevich)**.** *A set is Diophantine if and only if it is recursively enumerable.*

Proof. A Diophantine set is recursively enumerable, because we can try all the possible solutions and test them in order. The other direction is hard, but here is an overview. Let S be a recursively enumerable set. This means that S can be enumerated by a deterministic Turing machine. Now I want to write down a Diophantine equation such that it a tuple is being outputted if and only if it is a solution.

- We first arithmetize register machines. A register machine is a machine that is equivalent to a Turing machine. It has a register (which is like the tape in a Turing machine) and command lines (which is like the transition function in a Turing machine). We assign variables for each register and line, and then write down the relations.
- Then we Diophantize these relations. Many of the relations are going to be of the form
$$r \preceq s$$
which are called **bit maskings**. Here, r and s are binary numbers, and we define $r \preceq s$ if $r_i \leq s_i$ for all i. We are going to turn this into an exponential relation, using Lucas's theorem. (If you have done enough problem solving in high school, this is a standard trick.) Then we are going to show that this is a Diophantine relation.

So we turn a Turing machine into a Diophantine equation. $\square$

5.2 Register machines

So let me define a register machine. There are finitely many registers, $R_1, \dots, R_r$, and they can store nonnegative integers, of arbitrary size. It comes with a finite (command) lines $L_1, L_2, \dots, L_l$, where each L_i looks like

$$\begin{aligned} &L_i : R_j \leftarrow R_j \pm 1 \quad \text{or} \\ &L_i : \text{GOTO } L_k \quad \text{or} \\ &L_i : \text{IF } R_j > 0 (\text{or} = 0) \text{ GOTO } L_k. \end{aligned}$$

We say that M computes $y = f(x)$ if we have $x = (x_1, \dots, x_n)$ in the registers at time $t = 0$, and when the program ends, the values stored at the register are $f(x) = (f_1(x), \dots, f_n(x))$.

So let us try to arithmetize this register machine. Let us say that $R_1, \dots, R_n$ are our registers, $L_1, \dots, L_l$ are the lines, and $x \in \mathbb{N}^n$ is the input, with s the running time. First choose $Q = 2^{\text{big}}$ really big so that

$$x + s < \frac{Q}{2}, \quad l < \frac{Q}{2}, \quad r_{j,t} < \frac{Q}{2}.$$

This is going to be the possible range of the registers. Define the variables

$$\begin{aligned} r_{j,t} &= \text{register value of } R_j \text{ at time } t, \\ l_{i,t} &= \begin{cases} 1 & \text{machine carries out } L_i \text{ at time } t, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Then we define

$$R[j] = \sum_{t=0}^{s} r_{j,t} Q^t, \quad L[i] = \sum_{t=0}^{s} l_{i,t} Q^t$$

to make the data into a single number. Now we have the parameters x, y and variables $s, Q, R[1], \dots, R[n], L[1], \dots, L[l]$.

What are now the relations?

- start and end: $L_1 \succeq 1$ and $L_l = Q^s$,
- $Q = 2^t$,
- $x + s < Q/2$, $l < Q/2$, $R_j \preceq (Q/2 - 1)I$ (this enforces $r_{j,t} < Q/2$ because $r_{j,t}$ moves by at most 1),
- $I = (Q^{s+1} - 1)/(Q - 1)$,
- $L_i \preceq I$ and $\sum_{i=1}^{l} L_i = I$,
- execution commands: if $L_i : R_j \leftarrow R_j \pm 1$, then $QL_i \preceq L_{i+1}$, and other commands.

6 September 24, 2018

Last time we looked at register machines, which had registers $R_1, \ldots, R_r$ that can store arbitrarily large integers, and lines $L_1, \ldots, L_l$ that can change the value of a register by 1 or transfer to another line.

Example 6.1. Suppose you want to compute $f(x) = 2x$, and let's say that we start with x in R_2 and 0 in R_1. Then the register machine

L1 If $R_2 = 0$ Goto L_6

L2 $R_2 \leftarrow R_2 - 1$

L3 $R_1 \leftarrow R_1 + 1$

L4 $R_1 \leftarrow R_1 + 1$

L5 Goto L_1

L6 End

computes this.

Consider the function

$$G(l) = \max\{\text{output of a } l\text{-line machine with input } R_1 = 0\}.$$

This is well-defined, because there are only finitely many machines with l lines, up to equivalence. Suppose M is a c-line machine that computes $f(x)$. Then if we put x lines saying $R_1 \leftarrow R_1 + 1$ and then 5 lines $x \mapsto 2x$ and then c lines for M, we can compute $f(2x)$. So we get

$$f(2x) \leq G(x + 5 + c).$$

In particular, we can never compute G, because then $G(2x) \leq G(x + 5 + c)$ is a contradiction.

6.1 Register equations and their Diophantization

Given a register machine M, we want to find a polynomial $P(x, y, \ldots) = 0$ which has a solutions if and only if $y = M(x)$. We started with these variables

$$s, \quad r_{jt}, \quad l_{it}$$

as in the case of SAT. But then, the problem is that the number of variable depends on s. So instead, we defined $Q = 2^N$ and

$$R_j = \sum r_{jt} Q^t, \quad L_i = \sum l_{it} Q^t.$$

Then we had all these relations between $R_j, L_i, s, x, y, Q, I = \sum Q^t$. We could also recover r_{jt} and l_{it} by looking at the Q-ary expansion of R_j and L_i.

There were the universal equations, and the execution commands are the following:

- $QL_i \preceq L_{i+1}$ for L_i not containing Goto,
- $QL_i \preceq L_{i+1}+L_k$ and $QL_i \preceq L_k+(IQ-2R_j)$ (this requires some analysis), if L_i contains "If $R_j > 0$ goto L_k",
- something like $R_j = QR_j + \sum_i L_i - \sum_i L_i$ and $R_1 + yQ^s = R_1 Q + \sum_i L_i - \sum_i L_i + x$ that encodes how the register values transform.

So the point is that all of them are of the form (up to Diophantine relations)

$$a = b^c \text{ or } a \preceq b.$$

For the bit masking relation, we use the following theorem.

Theorem 6.2 (Lucas)**.** *If p is a prime, we have*

$$\binom{r}{s} \equiv \prod_i \binom{r_i}{s_i} \pmod p$$

where $r = \sum r_i p^i$ and $s = \sum s_i p^i$ are the p-ary expansions.

As a consequence, $s \preceq r$ is equivalent to $\binom{r}{s}$ being odd. Then this relation will be Diophantine if and only if I can encode $u = \binom{r}{s}$ as a Diophantine equation.

Theorem 6.3. *The set $\{(u,r,s) : u = \binom{r}{s}\}$ is Diophantine.*

Proof. We note that

$$\frac{(a+1)^r}{a^s} = a^{r-s} + \binom{n}{n-1} a^{r-s-1} + \cdots + \binom{r}{s} + \binom{r}{s-1}\frac{1}{a} + \cdots + \frac{1}{a^s}.$$

But we note that if $a > 2^r$, then the terms involving $\frac{1}{a}$ will sum to a number smaller than 1. This shows that for any $a > 2^r$, then

$$\mathrm{Rem}\left(\left\lfloor \frac{(a+1)^r}{a^s} \right\rfloor, a\right) = \binom{r}{s}.$$

Note that the relation $\mathrm{Rem}(b,a) = r$ is Diophantine, and similarly the integer part is also Diophantine. So we prove this theorem if we can encode the relation $a^b = c$. □

So everything reduces to the exponential relation.

Theorem 6.4. *The set $\{(a,b,c) : a = b^c\}$ is Diophantine.*

This uses Pell's equations, and is rather involved. I will only give an overview of how this works next time.

7 September 26, 2018

Last time wrote down the register relations, universal ones and program-specific ones. Many of them were bit-masking relations, and we reduced these to exponential relations. So we needed to know how we can encode the exponential relation

$$\{(a, b, c) : a^b = c\}.$$

This is what we are going to do today.

7.1 Diophantization of the exponential relation

Definition 7.1. For $d = a^2 - 1$ and a an integer, **Pell's equation** is the equation

$$x^2 - dy^2 = 1.$$

The equation admit solutions of the form

$$x_a(n) + y_a(n)\sqrt{a^2 - 1} = (a + \sqrt{a^2 - 1})^n.$$

Using this, we can prove that

$$\{(a, b, n) : b = x_a(n)\}$$

is Diophantine. In fact, the relation $c = y_a(b)$ can be encoded by

- $d^2 - (a^2 - 1)c^2 = 1$,
- $f^2 - (a^2 - 1)e^2 = 1$,
- $i^2 - (g^2 - 1)h^2 = 1$,
- $e = (i + 1)2c^2$,
- $g \equiv a \pmod{f}$,
- $g \equiv 1 \pmod{2c}$,
- $k \equiv c \pmod{f}$,
- $k \equiv b \pmod{2c}$,
- $b \leq 2c$.

To show this, let $h = y_g(r)$ for some r. Then show $b \equiv r \pmod{2c}$ and $r \equiv \pm p \pmod{2c}$, where $c = y_a(p)$. Then we can show that $b = p$ by using $b \leq 2c$.

Note that $x_a(n)$ and $y_a(n)$ grows exponentially in n. One can show that we have

$$(2a - 1)^n \leq y_a(n + 1) \leq (2a)^n.$$

Theorem 7.2 (Robinson)**.** *For all $n \geq 0$ and $b \geq 0$, we have*

$$x_a(n) - (a - b)y_a(n) \equiv b^n \pmod{2ab - b^2 - 1}.$$

Proof. I don't have any intuition for this, but you can play around with numbers. □

So if $a > y_b(n+1)$ then we have

$$b^n = \mathrm{Rem}(x_a(n) - (a-b)y_a(n), 2ab - b^2 - 1).$$

This is because $b^n < 2ab - b^2 - 1$ since a is really big. This finally shows that the exponential relation is Diophantine.

7.2 Finishing Hilbert's tenth problem

Theorem 7.3. *Hilbert's tenth problem is undecidable.*

Proof. Consider $S = L(\mathsf{HALT})$, which is undecidable but semidecidable. (This means that there is a register machine M such that $S = \{M(1), M(2), \dots\}$.) Suppose the problem is decidable. Then associated to M, there is a Diophantine equation such that

$$y = M(n) \quad \Leftrightarrow \quad \exists \vec{x}, P(y, n, \vec{x}) = 0.$$

So given y, we can test if $P(y, -, -) = 0$ has a solution by a register machine. This determines whether $y \in S$ or not. This contradicts that S is not decidable. □

Actually, we have a stronger statement. There exists a single (family of) Diophantine equation whose solvability cannot be algorithmically decided.

This whole proof implies that all computable functions are polynomials. Let me be more precise.

Proposition 7.4. *Let $y = f(x)$ be computable. Then there exists a polynomial $P(x, x_0, x_1, \dots, x_n)$ such that*

$$\{(x, y) : y = f(x)\} = \{(x, y) : \exists x_0, \dots, x_n, y = P(x, x_0, \dots, x_n)\}.$$

Proof. Because $\{y = f(x)\}$ is Diophantine, there exists a polynomial $Q(x, y, x_1, \dots, x_n)$ such that $y = f(x)$ if and only if $Q(x, y, x_1, \dots, x_n)$ for some x_i. This is then equivalent to existence of $x_0, \dots, x_n$ such that

$$(x_0 + 1)(1 - Q(x, x_0, x_1, \dots, x_n)^2) = y + 1.$$

This is called Putnam's trick. □

Also, we see that there exists a **universal Diophantine equation**.

Theorem 7.5. *Fix $n \in \mathbb{N}$. Then there exists a polynomial*

$$U_n(a_1, \dots, a_n, k, \underline{y})$$

such that for any polynomial $D(a_1, \dots, a_n, \underline{y})$, there exists a k_D such that

$$\{a : \exists \underline{x}, D(a, \underline{x}) = 0\} = \{a : \exists \underline{y}, U(a, k_D, \underline{y}) = 0\}.$$

Proof. We note that the Diophantine sets are enumerable, so let $S_1, S_2, \ldots$ be the sets. Let $M_1, M_2, \ldots$ be the machines enumerating the solutions, i.e., $S_i = \{M_i(1), M_i(2), \ldots\}$. Then we can construct a machine that enumerates

$$\{(a, k) : a \in S_k\},$$

by using the machines. So this is semi-decidable. The Diophantine equation associated to this is going to be the universal equation. $\square$

Index

bit masking, 12

clause, 8
computation path, 4

decidable set, 11
decision problem, 2
Diophantine, 10

language, 2
literal, 8

non-deterministic Turing machine, 4
NP-complete, 5
NP-hard, 5

Pell's equation, 16
polynomial time, 4

recursively enumerable set, 11
reduction, 5

semi-decidable, 11

time complexity, 3, 4
truth assignment, 8
Turing machine, 2

universal Diophantine equation, 17
universal Turing machine, 6