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

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. □

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 $C = \{U_1, \dots, U_l\}$ 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 $\{0 = q_0, 1 = q_Y, 2 = q_N, \dots, l\}$. 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. □

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 $f_1, \dots, f_m \in k[x_1, \dots, k_n]$, 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 $f_1, \dots, f_m \in k[x]$, 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 $S = \{x_1, x_2, \dots\}$. 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. □

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 $s, Q, R[1], \ldots, R[n], L[1], \ldots, L[l]$.

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 $R_1, \dots, R_r$ that can store arbitrarily large integers, and lines $L_1, \dots, L_l$ 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

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

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

6.1 Register equations and their Diophantization

Given a register machine M, we want to find a polynomial $P(x, y, \dots) = 0$ 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

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

Note that the relation $\operatorname{Rem}(b, a) = r$ 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. □

8 October 10, 2018

This was a guest lecture by Matthias Christandl. I will talk about quantum mechanics, applied to computer science. This was developed in the early 20th century, in order to overcome the difficulty of describing small particles. It is a mathematical theory that was hugely successful in both predicting and explaining new phenomena.

8.1 Crash course on quantum mechanics

There are some axioms.

- There is a complex Hilbert space $\mathcal{H}$, with a Hermitian metric, the system of the physics. Two systems $\mathcal{H}_A$ and $\mathcal{H}_B$ combine to $\mathcal{H}_A \otimes \mathcal{H}_B$.
- The state of a system is given by a vector $\psi \in \mathcal{H}$, normalized so that $\|\psi\| = 1$.
- The time evolution is given by the Schrödinger equation

  $$i\frac{\partial}{\partial t}\psi(t) = H(t)\psi(t),$$

  where $H(t)$ is the Hamiltonian (or energy) that is a Hermitian operator on $\mathcal{H}$.
- Measurement is given by $\{P_i\}_{i\in I}$, a family of projection, such that $\sum_{i\in I} P_i = \mathbf{1}$ and $P_iP_j = 0$ for $i \neq j$. If a measurement is carried out, you get the outcome "i" with probability

  $$p_i = \langle \psi, P_i\psi \rangle.$$

  After the measurement, the state becomes $\psi_i = \frac{1}{\sqrt{p_i}}P_i\psi$.

Example 8.1. There is the qubit, or the spin-$\frac{1}{2}$ system. This is the simplest possible non-trivial example, $\mathcal{H} = \mathbb{C}^2$. We put the inner product

$$\langle \psi, \phi \rangle = \bar{\psi}_0\phi_0 + \bar{\psi}_1\phi_1.$$

Let us look at the Hamiltonian

$$H(t) = H = \begin{pmatrix} 1 * 0 & \\ 0 & -1 \end{pmatrix}.$$

Then the time evolution of an arbitrary ψ will be

$$\psi \mapsto \psi_0 e^{-it}e_0 + \psi_1 e^{it}e_1.$$

There is a resolution of the identity,

$$P_0 = e_0e_0^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \quad P_1 = e_1e_1^\dagger \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Now let us imagine that we have a different apparatus, given by a different resolution of the identity

$$Q_0 = \frac{1}{2}(e_0 + e_1)(e_0 + e_1)^\dagger, \quad Q_1 = \frac{1}{2}(e_0 - e_1)(e_0 - e_1)^\dagger.$$

If you take this state $\frac{1}{\sqrt{2}}(e_0 + e_1)$, then you always get $Q = 0$ with probability 1. But if you measure it with P and then measure it with Q, the result will be $Q = 0$ with probability $\frac{1}{2}$ and $Q = 1$ with probability $\frac{1}{2}$.

This really is a generalization of the classical bit. You can also change the first and second components by unitary operators.

Example 8.2. For the harmonic oscillator, we have

$$\mathcal{H} = L^2(\mathbb{R}), \quad H = \frac{\omega^2 \hat{x}^2}{2} + \frac{\hat{p}^2}{2m},$$

where $\hat{x}\psi(x) = x\psi(x)$ and $\hat{p}\psi(x) = -i\frac{\partial}{\partial x}\psi(x)$. Then

$$H = \sum_n \Big(n + \frac{1}{2}\Big) f_n f_n^\dagger.$$

If we now have n qubits, the Hilbert space is

$$\mathcal{H} = \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2 \cong \mathbb{C}^{2^n}.$$

The basis of given by

$$e_{i_1} \otimes e_{i_2} \otimes \cdots \otimes e_{i_n}.$$

If we write $|e_0\rangle = |0\rangle$ and $|e_1\rangle = |1\rangle$, we can write this vector as $|i_1 i_2 \cdots i_n\rangle$. Then we can have states like

$$\psi = \frac{1}{\sqrt{2}}(|00\cdots 0\rangle + |11\cdots 1\rangle).$$

These are pretty hard to create in labs, for n pretty large. A quantum circuit is basically applying unitary operations one after another.

9 October 15, 2018

Today we will continue with classical boolean circuits. Then we will talk about quantum circuits and the Grover search algorithm.

Let us fix notation

$$\mathbb{B} = \{0, 1\}, \quad \mathcal{B} = \mathbb{C}^2,$$

the state spaces for the classical and quantum bits. The two vector $|0\rangle, |1\rangle$ are going to be orthonormal bases of the space $\mathcal{B}$. Then in

$$\mathcal{B}^{\otimes n} = \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2,$$

we can declare the vectors

$$|j\rangle = |j_1 j_2 \cdots j_n\rangle = |j_1\rangle \otimes \cdots \otimes |j_n\rangle$$

for $j_i \in \mathbb{B}$ to be orthonormal. Each "tensor factor" of $\mathcal{B}^n$ is called a single qubit.

Roughly, what a quantum computer can do is to

- perform unitary transformation operations, and
- perform measurements.

How can we use this to compute a boolean function, say something that an ordinary computer can do? There is the **Hadamard gate**

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

and then we can take

$$H^{\otimes n} |0^n\rangle = \frac{1}{2^{n/2}} \sum_{x \in \mathbb{B}^n} |x\rangle.$$

Then when we measure, we get each $|x\rangle$ with probability $1/2^n$.

To compute $f : \mathbb{B}^n \to \mathbb{B}^m$, we need to devise a unitary operator U such that given $x \in \mathbb{B}^n$,

(1) $U|x\rangle = \sum c_y |y\rangle$,

(2) $|c_y|^2$ is largest at $y = f(x)$.

9.1 Grover search algorithm

Here is the problem. There is a hidden $y_0 \in \mathbb{B}^n$, and we need to find y_0 by asking some "oracle" the question "Is y the hidden y_0?". Classically, we need to just guess $N = 2^n$ times and ask the queries. But in the context of quantum computation, we can do this in $\sqrt{N}$ queries with high probability.

In the quantum world, this is a unitary matrix U, given by

$$U|y_0\rangle = |y_0\rangle, \quad U|x\rangle = -|x\rangle \text{ for } x \neq y_0.$$

Define

$$V = I - 2|\xi\rangle\langle\xi|, \quad |\xi\rangle = \frac{1}{\sqrt{N}}\sum_x |x\rangle,$$

which is reflection with respect to hyperplane perpendicular to $|\xi\rangle$.

If you think about it, U is a linear combination of I and $|y_0\rangle\langle y_0|$. So the operator VU is a rotation in the plane spanned by y_0 and ξ. But here, note that

$$\sin\varphi = \langle\xi|y_0\rangle = \frac{1}{\sqrt{N}}.$$

Here, you have to do some analysis. Each time we apply VU, the rotation is by 2φ. So when we do the rotation m times, to get

$$(VU)^m|\xi\rangle,$$

the rotation angle is $m2/\sqrt{N} \approx \frac{\pi}{2}$.

But how would you implement this algorithm? We certainly can't build a machine for each unitary operator. So we want a relatively small set of unitary operators, and simulate other operators using the basic ones. We will show that any unitary operator can be approximated by a "circuit" over a finite set of unitary operators.

9.2 Quantum circuits

Definition 9.1. A Boolean function is a map $f : \mathbb{B}^n \to \mathbb{B}$, and a **boolean circuit** (over $\mathcal{A}$) is a representation of a boolean function as a composition of other boolean functions. It consists of

- variables $x_1, x_2, \dots, x_n$,
- auxiliary variables $y_1, y_2, \dots, y_m$,
- assignments $y_i = f_j(x, y_1, \dots, y_{j-1})$, where $f_j \in \mathcal{A}$.

You can also think of it as an acyclic directed graph with input vertices with in-degree 0 and output vertices with out-degree 0. There are gates G each with out-degree 1 and a map $G \to \mathcal{A}$.

Theorem 9.2. *Any boolean function can be computed by a circuit over*

$$\mathcal{A} = \{\wedge, \vee, \neg\}.$$

A **quantum circuit** is the same thing, but with input qubits, and gates unitary operators.

Lemma 9.3. *Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be computed by a (boolean) circuit of size L over $\mathcal{A}$. Then a map of the form*

$$(x, 0^l) \mapsto (f(x), g(x))$$

can be computed by a circuit of size $O(L)$ over the set $\{h_\oplus : h \in \mathcal{A}\} \cup \{\mathrm{id}_\oplus\}$, where this $\oplus$ are the invertible maps

$$h_\oplus(x, y) = (x, y \oplus h(x)).$$

So we are introducing these ancilla qubits.

Theorem 9.4. *Any unitary operator admits an exact realization over $\mathcal{G}_1 \amalg \mathcal{G}_2$, where $\mathcal{G}_i$ is the set of all quantum gates with i qubit inputs.*

This only says that any unitary matrix can be decomposed into 2×2 unitary matrices.

Theorem 9.5. *There is the approximate basis, called the* ***standard quantum basis***

$$Q = \{H, \sigma^x, K^{\pm 1}, \Lambda\sigma^x, \Lambda^2\sigma^x\},$$

that generates a dense subgroup of $\mathcal{U}(\mathcal{B}^{\otimes 3})$.

10 October 17, 2018

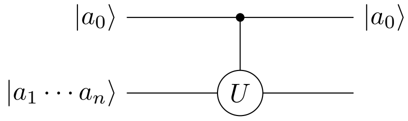
Recall that we set $\mathbb{B} = \{0, 1\}$ and $\mathcal{B} = \mathbb{C}^2$. A quantum gate is just a unitary transformation, and a quantum circuit is just the same as a boolean circuit but with qubit input and quantum gates. We saw the example of the Grover search algorithm, which is just finding an answer by quering the oracle. It turned out that only $\approx 2^{n/2}$ queries are needed to find the answer.

10.1 Circuit for the Grover search algorithm

Definition 10.1. Let U be a quantum gate, i.e., $U : \mathcal{B}^{\otimes n} \to \mathcal{B}^{\otimes n}$. Then we can define

$$\Lambda U : \mathcal{B}^{\otimes(n+1)} \to \mathcal{B}^{\otimes(n+1)}; \quad |a_0 a_1 \cdots a_n\rangle \mapsto \begin{cases} |a_0 a_1 \cdots a_n\rangle & \text{if } a_0 = 0, \\ |a_0\rangle \otimes U|a_1 \cdots a_n\rangle & \text{if } a_0 = 1. \end{cases}$$

We are going to use the circuit diagram



to represent this.

Now we can use this to build a quantum circuit for the Grover search algorithm. It suffices to build a quantum circuit for $V = I - 2|\xi\rangle\langle\xi|$, because the gate U is already given to us. But then, we just have $|\xi\rangle = H^{\otimes n}$. So it really suffices to build the circuit for

$$I - 2|0^n\rangle\langle 0^n|.$$

Define the function

$$F : \mathbb{B}^{n+1} \to \mathbb{B}^{n+1}; \quad (a_0, \dots, a_n) \mapsto (a_0 \oplus (\neg(a_1 \vee \cdots \vee a_n)), a_1, \dots, a_n).$$

Note that F is a permutation.

Proposition 10.2. *Any permutation can be realized as a circuit over $\{\neg, \wedge_\oplus\}$ using ancillas.*

Corollary 10.3. *For any permutation F, its associated quantum gate $\hat{F}$ can be realized as a circuit over $\{\hat{\neg}, \hat{\Lambda}_\oplus\}$ using ancillas.*

Here, the quantum analogues $\hat{\neg}$ and $\hat{\Lambda}_\oplus$ are just σ_x and $\Lambda^2\sigma_x$. So they are in our set of standard quantum gates.

Let this circuit (with ancillas) be Z. Now consider the circuit in Figure 1. Then if we feed in $|0, 0^n, 0^l\rangle$, we get

$$Z|0, 0^n, 0^l\rangle = |F(0, 0^n), G\rangle = |1, 0^n, G\rangle,$$

and then σ_z will turn it into

$$-|1, 0^n, G\rangle,$$

and so applying Z again will give

$$Z(-|1, 0^n, G\rangle) = -|0, 0^n, 0^l\rangle.$$

So this does exactly what we wanted it to do.

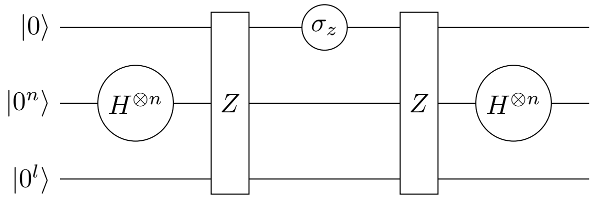


Figure 1: The quantum circuit for Grover's search algorithm

10.2 Quantum Fourier transform

Classical fast Fourier transform takes $O(n2^n)$, but the quantum Fourier transform only takes $O(n^2)$ time.

Definition 10.4. The **discrete Fourier transform** is the linear map

$$(x_0, \dots, x_{N-1}) \in \mathbb{C}^N \mapsto (y_0, \dots, y_{N-1}); \quad y_k = \frac{1}{\sqrt{N}} e^{2\pi ijk/N} x_j.$$

If we do this naïvely, we are doing multiplication and addition $O(N^2)$ times. In fast Fourier transform, we employ divide and conquer to bring it down to $O(N \log N)$ operations. If N is even, we divide

$$\sum e^{2\pi ijk/N} x_j = \sum e^{2\pi i(2m)k/N} x_{2m} + \sum e^{2\pi i(2m+1)k/N} x_{2m+1}.$$

So we don't have to compute them twice.

Definition 10.5. The **quantum Fourier transform** is defined as

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle.$$

(Note that this is a unitary operator!) We interpret

$$|j\rangle = |j_1 j_2 \cdots j_n\rangle$$

with the encryption $j = j_1 2^{n-1} + \cdots + j_{n-2} 2 + j_n$. We can also write $0.j_1 j_2 \cdots j_n = 2^{-n} j$.

We know that $\mathcal{B}^{\otimes n}$ has a standard orthonormal basis

$$\{|0\rangle, |1\rangle, \dots, |N-1\rangle\}$$

where $N = 2^n$.

Proposition 10.6. *Actually, the quantum Fourier transform is equivalent to*

$$|j\rangle \mapsto (|0\rangle + e^{2\pi i 0.j_n}|1\rangle) \otimes (|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle) \otimes \cdots \otimes (|0\rangle + e^{2\pi i} 0.j_1 j_2 \cdots j_n |1\rangle) \frac{1}{2^{n/2}}.$$

Proof. Note that $|j\rangle \mapsto \sum_k \omega^k |k\rangle$. But here,

$$\omega^k |k\rangle = \omega^{k_1 2^{n-1}} |k_1\rangle \otimes \cdots \otimes \omega^{k_n} |k_n\rangle$$

and note that

$$\omega^{k_s 2^{n-s}} = \begin{cases} |0\rangle & k_s = 0 \\ (e^{2\pi i 0.j_{n-s+1} \cdots j_n})|1\rangle & k_s = 0. \end{cases}$$

Then the formula follows. □

So how will we make this into a circuit? We have H, and we will use the gates

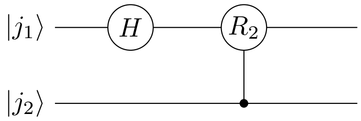
$$R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}$$

for each k. (If you really want to do over the standard gates, you approximate.)

First we note that we can conveniently write

$$H|j_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1}|1\rangle).$$

This is nice, because it is what we needed. For the next thing, we look at the following circuit:



Then we first get

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1}|1\rangle)|j_2\rangle$$

after H, and then we will get

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1 j_2}|1\rangle)|j_2\rangle.$$

11 October 22, 2018

Last time we talked about the quantum Fourier transform

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle.$$

There was a product representation

$$|j\rangle \mapsto 2^{n/2}(|0\rangle + e^{2\pi i 0.j_n}|1\rangle) \cdots (|0\rangle + e^{2\pi i 0.j_1 \cdots j_n}|1\rangle),$$

and it turned out that this made it easy to construct a quantum circuit for the problem. At the end, we get a quantum circuit of size $\Theta(n^2)$.

11.1 Phase estimation

Given a unitary operator U and an eigenvector $|u\rangle$, this question is about finding a ϕ such that $U|u\rangle = e^{2\pi i\phi}|u\rangle$. Using this, we can do "order finding" pretty efficiently. This is about, given a composite N and $1 < a < N$, finding a smallest positive r such that $a^r \equiv 1 \pmod{N}$.

Going back to phase estimation, consider a unitary operator U and $|u\rangle$ an eigenvector U. Assume that

(1) we have a quantum computer that can set a register at $|u\rangle$, (this is not really obvious; this state $|u\rangle$ can be a complicated superposition)

(2) we can compute $\wedge U^{2^j}$,

(3) the eigenvalue takes the form of $\varphi = 0.\varphi_1\varphi_2 \cdots \varphi_t$.

So here is the circuit in Figure 2.

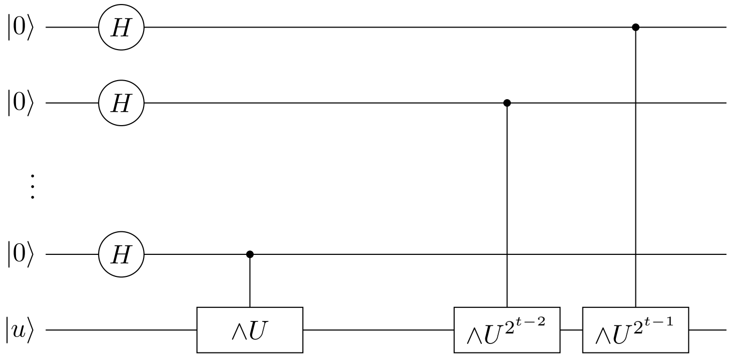


Figure 2: Quantum circuit for phase estimation

What does this do? Note that

$$U^{2^s} = (e^{2\pi i 0.\varphi_1 \cdots \varphi_t})^{2^s}|u\rangle = e^{2\pi i 0.\varphi_{s+1} \cdots \varphi_t}|u\rangle.$$

So if we do $\wedge U^{2^{t-1}}$, we get something like

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.\varphi_t}|1\rangle)|u\rangle.$$

If we analyze the what the other parts are doing, we see that end result is going to be

$$(\text{quantum Fourier transform of } |\varphi\rangle)|u\rangle.$$

Now we take the first qubits, and then apply the inverse Fourier transform. (This can be done by just reversing the circuit for the Fourier transform.) Then we recover $|\varphi\rangle$.

11.2 Order finding

Let $N = \prod_{j=1}^{k} p_j^{\alpha_j}$ be a composite number, assume that we are given $1 < x < N$. We want to find an r such that $x^r \equiv 1 \pmod{N}$. The idea is to sample enough points, apply quantum Fourier transform, and observe.

We will sample q points, where $N^2 < q \leq 2N^2$. Let us assume that there is an efficient quantum circuit for computing

$$F : |a, 0\rangle \mapsto |a, x^a \bmod N\rangle.$$

There is a classical algorithm that does this in pretty efficient time, about $O((\log_2 N)^2)$. So this also can be done using a quantum algorithm, using ancilla bits.

Now here is what we do. We start with $|0, 0\rangle$, and first apply quantum Fourier transform to the first register, so that we get

$$\frac{1}{\sqrt{q}} \sum_{k=0}^{q-1} |k, 0\rangle.$$

Now we apply F to this state and get

$$\frac{1}{\sqrt{q}} \sum_{k=0}^{q-1} |k, x^k \bmod N\rangle.$$

But then we see that this second component is periodic with period r. So when observe that the second state is. Then the state collapses to something like

$$\frac{1}{\sqrt{q/r}} \sum_{j=0}^{q/r-1} |jr + s\rangle(\otimes|x^s\rangle)$$

for some s.

Now we take the quantum Fourier transform and measure the state. We expect this to give something like multiples of q/r. In fact, with high probability we will get a k such that

$$\left|\frac{b}{r} - \frac{k}{q}\right| < \frac{1}{2q}.$$

So we can just use continued fractions for $\frac{k}{q}$ to recover r. Here, there is a problem if b happens cancel out a lot of factors of r. But b is chosen randomly modulo r, so if we do this enough times (say $q/\phi(q) = O(\log\log q)$ times) then we will get in a situation where $\gcd(b, r) = 1$. Then we exactly get r.

This can be used to do integer factorization. We input $y > 1$ a positive integer and want to factorize y. Here is the classical algorithm:

(1) If y is even, just output 2.

(2) Check whether $y = m^k$, and if yes, output m. (You try for k up to $\log_2 y$.)

(3) Randomly choose $a \in \{0, \cdots, y-1\}$ and compute $\gcd(a, y) = b$. If $b > 1$, output b.

(4) Compute $n = \mathrm{ord}_y(a)$, and if r is odd, output that it is prime.

(5) Compute $d = \gcd(a^{r/2} - 1, y)$, and if $d > 1$ output d and if $p = 1$ output prime.

You can show that

$$\Pr(\text{algorithm returns prime} \mid y \text{ composite}) < \frac{1}{2}.$$

So repeating the algorithm gives an accurate result.

12 October 24, 2018

Last time we looked at quantum phase estimations, which was finding the eigenvalue of

$$U|u\rangle = e^{2\pi i\varphi}|u\rangle.$$

We also looked at how to find the order of an element $1 < x < N$ modulo N. We first make this state $\frac{1}{\sqrt{q}}\sum|a, x^a\rangle$, and then measure to collapse the state. Then we take the quantum Fourier transform to extract the period.

We had an algorithm of factorizing an integer, if we know how to find the order.

(1-3) Checks if a given integer y is even or a perfect power. Then chooses a random a and computes $d = \gcd(a, y)$.

(4) Computes $m = \operatorname{ord}_y(a)$ and if m is odd, returns "prime".

(5) Computes $\gcd(a^{m/2} - 1, y) = b$ and if $b = 1$, returns "prime".

This algorithm is correct with positive probability because of the following fact.

Lemma 12.1. *Let $y = \prod_j p_j^{\alpha_j}$ and let $m_j = \operatorname{ord}_{p_j^{\alpha_j}}(a) = 2^{s_j}r_j$. Then the algorithm returns "prime" if and only if $s_1 = \cdots = s_k$.*

So the probability that the algorithm returns "prime" when n is not prime is equal to the probability that $s_1 = \cdots = s_k$. You can estimate this, and it is smaller than $\frac{1}{2}$. This means that we can run the algorithm repeatedly many times and get this.

12.1 Complexity class BQP

I've been conflating quantum algorithms and quantum circuits. So we can just define complexity using this.

Definition 12.2. We define the **complexity** of a quantum algorithm as the quantum circuit size. (Here, we assume that every gate computes in unit cost.)

Suppose we want to compute

$$f : \mathbb{B}^* \to \mathbb{B}^*.$$

What does it mean for a quantum algorithm to compute f, where $\mathbb{B}^* = \coprod_n \mathbb{B}^n$.

Definition 12.3. A **quantum algorithm** for computing f is a Turing machine M that returns on input n, a quantum circuit Z_n that computes $f|_{\mathbb{B}^n}$.

We view quantum algorithms as a discrete time model, but with continuous space. Then we define the **quantum complexity** of M as the complexity of M as a Turing machine. Here, "returns" means that it describes all the circuit diagram and all the entries of the gates (with given precision) and so on. So the quantum complexity already encodes the circuit size of the output.

Definition 12.4. A language L is said to be *BQP* if there is a quantum algorithm M for L with polynomial complexity.

12.2 The Blum–Shub–Smale model

Computation on $\{0, 1\}$ is good for formal logic, but it is not suitable for mathematical analysis. So we want a model more suitable for complexity of numerical analysis. Roughly, a Blum–Shub–Smale machine is a machine with registers that can store arbitrary real numbers and can perform basic arithmetic operations, and also branch according to register values.

Example 12.5. A nice example is Newton's method. We are given f and ϵ, a function and precision. Given an input z, we iterate

$$z \leftarrow N_f(z) = z - \frac{f(z)}{f'(z)}$$

until we get $|f(z)| < \epsilon$.

Definition 12.6. A **Blum–Shub–Smale machine** consists of

(1) a finite directed connected labeled graph with five types of nodes:

- an input node (unique),
- an output node,
- a computation node (with one subsequent nodes),
- a branch node (with two subsequent nodes, with edges labeled $+$ and $-$),
- a shift node (with one subsequent nodes),

(2) the input space $\mathcal{I}_M$, the output state $\mathcal{O}_M$, and the state space $\mathcal{S}_M$, where the computation occurs,

(3) associations to each nodes a (polynomial) map

- input - $I : \mathcal{I}_M \to \mathcal{S}_M$,
- output - $O : \mathcal{S}_M \to \mathcal{O}_M$,
- computation - $f : \mathcal{S}_M \to \mathcal{S}_M$,
- shift - $\sigma_l, \sigma_r : \mathcal{S}_M \to \mathcal{S}_M$,
- branch - $h : \mathcal{S}_M \to R$ for some ring R (like $\mathbb{R}$ or $\mathbb{C}$ or $\mathbb{F}_2$).

What the machine does is just to follow the flowchart. Define

$$R^\infty = \coprod_n R^n, \quad R_\infty = \{(\dots, a_{-1}, a_0, a_1, \dots) : a_i \in R\}.$$

We can take $\mathcal{I}_M = \mathcal{O}_M = R^\infty$ and $\mathcal{S}_M = R_\infty$. (The computation power comes from the fact that state spaces are infinite.) Given $x \in \mathcal{S}_M$, we compute $I(x) \in \mathcal{S}_M$. If it is a computation node, just do the computation, update the state, and move to the next node.

13 October 29, 2018

So a BSS machine was a finite directed labaled graph with nodes input, output, computation, branch, shift, spaces $\mathcal{I}_M$, $\mathcal{O}_M$, $\mathcal{S}_M$, and polynomial maps indicating the computation.

13.1 Examples of Blum–Shub–Smale machines

We used the spaces

$$R^\infty = \bigcup R^n, \quad R_\infty = \{(\dots, a_{-1}, a_0, a_1, \dots)\}.$$

Usually, I and O are usually the linear maps

$$\begin{aligned} I &: (x_1, \dots, x_n) \mapsto (\dots, 0, 1, \dots, 1, a_1 = x_1, \dots, x_n, 0, \dots), \\ O &: (\dots, x_{-1}, x_0, x_1, \dots) \mapsto (x_1, \dots, x_l) \end{aligned}$$

because otherwise we need to know where the input ends. Also, the branch an computation maps are polynomials (that is, each entry is a polynomial). Here, we only allow polynomials to have finite **dimension** and **degree**. This means that if we write

$$g = (\dots, g_0, g_1, \dots) : R_\infty \to R_\infty,$$

then we have

$$g_i(x) = x_i \text{ for } i \leq 0 \text{ and } i \geq n+1.$$

The dimension is this n, and the degree is the supremum of the degrees of g_i. To use the negatively graded numbers, we use shift maps.

Definition 13.1. We define $\sigma_l, \sigma_r : \mathcal{S}_M \to \mathcal{S}_M$ as

$$\sigma_l(x) = (\dots, x_{-1}, a_1 = x_0, x_1, \dots), \quad \sigma_r(x) = (\dots, x_1, a_1 = x_2, x_3, \dots).$$

Definition 13.2. If M is a BSS machine, we define

$$\begin{aligned} K_M &= \text{dimension of } M = \max\{\dim \eta : \eta \text{ a computation node}\}, \\ D_M &= \text{degree of } M = \max\{\deg \eta : \eta \text{ a computation node}\}. \end{aligned}$$

We can give an interpretation of the BSS machine that is similar to the Turing machine. Let $\mathcal{N}$ be the set of nodes, and let us denote $\mathcal{S} = \mathcal{S}_M$. Then we can define the **computing endomorphism** as

$$H(\eta, \xi) = (\beta_\eta, g_\eta(\xi)),$$

where β_η is the next node. This function has complete information of the computation of M on x. We can also consider this as a dynamical system.

Definition 13.3. A **dynamical system** is an action of a monoid G on X.

We can now define the **computation path** as the sequence

$$\gamma_x = \{\eta^k(x) = \pi_{\mathcal{N}} H^k(x, I(x))\}$$

of nodes, and similarly the **state trajectory** as the sequence

$$\{q^k(x) = \pi_{\mathcal{S}} H^k(x, I(x))\}.$$

We say that M halts on x if the output node some $\eta^T(x)$. Then the complexity of M can be written as

$$T_M(x) = \min\{T : \eta^T(x) \text{ is the output node}\}.$$

Example 13.4. Turing machines are BSS machines. A Turing machine has symbols $\{0, 1, \text{blank}\}$ with states $\{1 = q_{\text{No}}, 2 = q_{\text{Yes}}, 3 = q_0, 4, \dots, N\}$. We can take $R = \mathbb{F}_3 = \{0, 1, -1\}$ where we consider -1 as a separator. The graph is going be one node of each kind, input, computation plus shift, branch, output. Then we define

$$I(x_1, \dots, x_n) = (\dots, 0, -1, 1^n, -1, q_0, -1, x_1, x_2, \dots, x_n, 0, \dots).$$

and do something.

13.2 Decidability for BSS machines

We define

$$\mathcal{D}_M(T) = \{x \in \mathcal{I} : M \text{ halts on } x \text{ in time } T\},$$

and then define the halting set of M as

$$\mathcal{D}_M = \bigcup_{T=1}^{\infty} \mathcal{D}_M(T).$$

Then we can define the computation function as a partial function

$$\Phi_M : \mathcal{I}_M \to \mathcal{O}_M; \quad x \in \mathcal{D}_M(T) \mapsto O(q^T(x)).$$

Then for any f a partial function on $\mathcal{I}_M$, we say that M computes f if $\mathcal{D}_M \supseteq \text{Domain}(f)$ and $\Phi_M(x) = f(x)$ for all $x \in \text{Domain}(f)$.

Definition 13.5. We say that $S \subseteq R^n$ is decidable if there exists a BSS machine M over R that computes χ_S.

This is really the motivation for Blum–Shub–Smale to come up with this machine. The **Mandelbrot set** is defined as the set

$$\{c \in \mathbb{C} : c, c + c^2, (c + c^2)^2 + c, \dots \text{ is bounded}\}.$$

Theorem 13.6. *The Mandelbrot set is* not *decidable over* $\mathbb{C}$.

This follows from the analysis of the boundary of the Mandelbrot set. Still, the complement of the Mandelbrot set is semi-decidable.

Definition 13.7. A set $S \subseteq R^n$ is said to be **basic semi-algebraic** if it is defined by finitely many polynomial equations and inequalities. Then a **semi-algebraic set** set is a finite union of basic semi-algebraic sets.

Theorem 13.8 (path decomposition)**.** *Let M be a finite BSS machine over R. Then for any $T > 0$, the set $\mathcal{D}_M(T)$ is a countable union of semi-algebraic sets.*

14 October 31, 2018

We stated the path decomposition theorem last time.

Theorem 14.1 (path decomposition)**.** *Let M be a finite-dimensional machine. Then $\Omega_M = \{x \in \mathcal{I}_M : M \text{ halts on } x\}$ is a countable union of semi-algebraic sets.*

If you think about this, if we fix a computation path, the states are determined as polynomials, and the condition that the computation really does follow this path is given by a bunch of inequalities. So this is a basic semi-algebraic set, and then we are taking the union over all computation paths. Now it can be shown that the boundary of the Mandelbrot set has Hausdorff dimension 2, and this shows that it cannot be written as a union of countable semi-algebraic sets.

14.1 The class NP over rings

A language L in this context is going to be a subset of $\mathcal{I}_M$. A structured decision problem is a tower

$$X_{\text{yes}} \subseteq X \subseteq \mathcal{I}_M,$$

where there are some "no" instances $X \setminus X_{\text{yes}}$, and some "non" instances $\mathcal{I}_M \setminus X$. We can define the running time of a BSS machine, so we can define

$$\mathcal{P}_{/R} = \{(X, X_{\text{yes}}) : \text{exists polynomial BSS over } R \text{ deciding } X_{\text{yes}}\}.$$

Similarly, we can define

$$\mathcal{NP}_{/R} = \left\{ (X, X_{\text{yes}}) : \begin{array}{c} \text{exists polynomial BSS over } R \text{ taking two inputs} \\ \text{such that } x \in X_{\text{yes}} \text{ if and only if there is } y \text{ with} \\ y \leq \text{poly}(|x|) \text{ and } M \text{ accepting } (x, y) \end{array} \right\}.$$

Here are some $\mathcal{NP}$-problems.

- Hilbert's Nullstellensatz (HN): given $f_1, \dots, f_l \in R[x_1, \dots, x_n]$, decide if there exists a common zero in R^n.
- $4-\mathsf{FEAS}$: given a single $\deg 4$ polynomial f in $R[x_1, \dots, x_n]$, decide if there exists a zero of f in R^n.

Note that if we have HN, we can always reduce it to all the polynomials into quadratic polynomials. Then instead of asking if there is a zero for $f_1, \dots, f_l$, we can ask for the solution of $f = \sum_i f_i^2$. This is why $4-\mathsf{FEAS}$ is a reasonable problem to consider.

- $\mathsf{SA}-\mathsf{FEAS}$: given a semi-algebraic system (this is a formula like $\bigvee((\varphi_{i1} > / = / < 0) \wedge \cdots))$ decide if it has a solution in R^n.
- $\mathsf{QA}-\mathsf{FEAS}$: in the case when the ring is not ordered, replace $>$ by $\neq$.

Theorem 14.2. *The problem* $\mathsf{SA}-\mathsf{FEAS}$ *is* $\mathcal{NP}$*-complete over* R*.*

Proof. Let $X_{\text{yes}} \subseteq X$ be in $\mathcal{NP}$. We need a map $\phi : X \to \mathsf{SF}$ computable by a polynomial BSS machine. Consider M the polynomial BSS machine over R, taking two inputs, (x, y) that solves the $\mathcal{NP}$-problem. Now we are going to encode this using register equations. Let $\mathcal{N} = \{1, \dots, N\}$ be the nodes, with 1 the input and N the output. Let us assume that this uses coordinates $a_{-m}, \dots, a_m$. Then we can set variables $a_{j,t}$ as nonzero if M is at node i at time t, and denote the registers as $q^t = (q^t_{-m}, \dots, q^t_m)$ at time t. Then we can down the equations

- $a_{1,0} = 1$ and $q^0 = I(x) = \dots 0x0 \dots$ for initialization,
- $a_{N,T} = 1$ for halting,
- $(a_{\beta(j),t+1} = 1) \wedge (a_{j,t} = 1)$ for moving computation nodes,
- $a_{j,t}a_{j',t} = 0$ for all $j \neq j'$, for all t, for uniqueness of the node,
- $(a_{j,t} = 1) \wedge ((h(q^t) \geq 0 \wedge a_{\beta(j),t+1} = 1) \wedge \neg)$ for branch nodes,
- $(q^{t+1} = G_j(q^t)) \wedge (a_{j,t} = 1)$ for input and computation and output nodes, where G_j is the polynomial determined by the polynomials.

So this becomes a semi-algebraic system, and this is what we wanted. □

15 November 5, 2018

Last time we showed that $\mathsf{SA}-\mathsf{FEAS}$ is $\mathcal{NP}$-hard over $\mathbb{R}$. It is easy to see that the problem itself is $\mathcal{NP}$ is $\mathcal{NP}$, so it is $\mathcal{NP}$-complete. This is because we can use the guess as a possible solution.

Proposition 15.1. *Over $\mathbb{Z}$, we have $\mathcal{P} \neq \mathcal{NP}$.*

Proof. By the DPRM theorem, HALT is Diophantine. So this is $\mathcal{NP}$ over $\mathbb{Z}$. But this is not decidable over $\mathbb{Z}$, and so it is clearly not $\mathcal{P}$. This shows that $\mathcal{P} \neq \mathcal{NP}$. □

Also, universal BSS machine exists. This can be constructed in a manner similar to the classical one.

- Encode the BSS machine as a sequence of sequences $(\eta, \beta_\eta, g_\eta)$ in some way.
- Divide the state space into two components, one for storing the input and the other for the "workspace".
- Now design a universal polynomial evaluator, which reads a polynomial and a state and then computes the result.
- Copy the state input into the workspace.
- Using the universal evaluator, simulate the input BSS machine, and only use the workspace to store the state data.

15.1 Continuous time systems

We have seen Turing machines, BSS machines, quantum computers. These are all discrete-time systems; computation is done step-by-step. A **continuous time system** is a system associated with a continuous dynamical system. The most important ones are

- differential analyzers (Bush, 1931),
- GPAC, general purpose analog computer (Shannon, 1941).

One of the reasons these are important are there are neural networks and natural computation or so on. There is no unified theory of these computations.

This **GPAC** consists of families of circuits with constants, adders, multipliers, and integrators. In theory, GPACs are much more powerful than Turing machines. Turing machines can compute only polynomials (this was the DPRM theorem plus Putnam's trick). But in theory, GPAC can generate exponential functions, logarithmic functions, trigonometric functions, and so on.

Theorem 15.2. *A GPAC can compute $f(t)$ if and only if it is "differentiably algebraic", i.e., it satisfies an ODE of the form*

$$p(t, f, f', \dots, f^{(n)}) = 0$$

for p a polynomial.

So let us use a general ordinary differential equation

$$y' = f(t, y), \quad y(0) = y_0,$$

where y can possibly be a vector.

Theorem 15.3. *Any Turing machine can be written down as an ordinary differential equation.*

Let us write by ψ as a transition function of a deterministic Turing machine M. We first need to encode the state of the machine. If the machine is like

$$\cdots bba_{-l} \cdots a_{-1}a_0a_1 \cdots a_pbb \cdots,$$

where a_0 is the head position, define

$$y_1 = a_0 + 10a_1 + \cdots + 10^pa_p, \quad y_2 = 10^{-1}a_1 + \cdots + 10^la_l, \quad y_3 = q$$

where q is the machine state. Now we want to construct an ordinary differential equation $y' = f(t, y)$ whose solution $\vec{y}(t) = (y_1, y_2, y_3)$ is the machine at time $t \in \mathbb{N}$.

Lemma 15.4. *Given $0 < \epsilon < \frac{1}{2}$, and a map $\psi : L \to L$ for some $L \subseteq \mathbb{N}^3$, there exists an analytic extension h_M such that*

$$|x - \bar{x}|_\infty \leq \epsilon \quad \Rightarrow \quad |\psi^{[j]}(x) - h_M^{[j]}(\bar{x})|_\infty \leq \epsilon$$

for all $x \in L$ and $j \in \mathbb{N}$. ($\psi^{[j]}$ is the jth iteration.)

The proof is really complicated. But using this, we can prove the following.

Theorem 15.5. *There is an analytic map*

$$p_M : \mathbb{R}^{m+4} \to \mathbb{R}^{m+3}$$

and a constant $g_0 \in \mathbb{R}^m$ such that for all $x_0 \in \mathbb{N}^3$, the solution $z(t)$ to

$$z' = p_M(t, z), \quad z(0) = (x_0, y_0)$$

satisfies

$$\|z_1(j) - \psi^{[j]}(x_0)\|_\infty < \epsilon$$

for all $j \in \mathbb{N}$, where $z = (z_1, z_2)$.

Proof. Let us write

$$z_1' = (h_M(r(z_2)) - z_1)^3\phi_1(t), \quad z_2' = (r(z_1) - z_2)^3\phi_2(t),$$

where r is a smooth map approximating $r(x) = \lfloor x + \frac{1}{2} \rfloor$ and ϕ_1 and ϕ_2 are approximating $\phi_1 = \sum_n \chi_{[n, n+\frac{1}{2}]}$ and $\phi_2 = \sum_n \chi_{[n-\frac{1}{2}, n]}$. Then if t is between 0 and $\frac{1}{2}$, then $z_1' = 0$ and so we are solving something like

$$y' = c(b - y)^3\phi(t).$$

Then we can solve this as

$$\frac{dy}{(b-y)^3} = c\phi(t)dt, \quad (b-y)^2 \leq \frac{1}{c\int \phi}.$$

So $z_2 \to y$ around $t = \frac{1}{2}$. Then in the next time interval, we get $z_1 \to h_M(y)$, and then after that, $z_2 \to h_M(y)$, and then $z_1 \to h_M^2(y)$, and so on. So you keep applying h_M. □

16 November 7, 2018

Last time we talked about this universal polynomial evaluation map $(f, x) \mapsto f(x)$. We first fix a degree d and the number of variables n, then we can compute the polynomial from the input and the coefficients. This is because the output is a polynomial in terms of the input and the coefficients. But this is not necessarily true. In this case, we can use something like

$$(n, x) \mapsto (n, x, x) \mapsto (n-1, x, x^2) \mapsto (n-2, x, x^3) \mapsto \cdots \mapsto (1, x, x^n).$$

Using this universal polynomial evaluation (UPE) module, we can effectively determine if $x \in \mathbb{R}^n$ satisfies a given semi-algebraic system. This shows that $\mathsf{SA} - \mathsf{FEAS} \in \mathcal{NP}$ over R.

We were also talking about continuous time systems, given by ordinary differential equations $y' = f(t, y)$. The claim was that Turing machines can be made into continuous time systems. To do this, we encode the machine in three numbers,

$$y_1 = a_0 + 10a_1 + \cdots + 10^p a_p, \quad y_2 = a_{-1} + 10a_{-2} + \cdots + 10^{n-1}a_{-n+1}, \quad q.$$

Now we used this proposition that ψ admits an analytic extension $\psi : \mathbb{R}^3 \to \mathbb{R}^3$. Then we were looking at differential equations like

$$z' = c(b-z)^3\phi(t).$$

If we solve this, we get

$$\frac{dz}{(b-z)^3} = c\phi(t)dt, \quad z(1) \approx b$$

if c is large enough. So we couple these equations and get

$$z_1' = c(f(r(z_1)) - z_1)^3\phi_1, \quad z_2' = c(r(z_1) - z_2)^3\phi_2$$

where ϕ_1 is like $\sum_n \chi_{[n+1/2,n+1]}$ and ϕ_2 is like $\sum_n \chi_{[n,n+1/2]}$. So what ends up happening is that z_2 copies z_1 and z_1 computes $f(z_2)$.

16.1 Simulating Turing machines over a polynomial ODE

This is clearly not analytic, since there are completely flat regions. So here, ϕ_1 and ϕ_2 should be analytic, so they are not entirely zero on this half-intervals, but we will bring ϕ_1 and ϕ_2 very close to zero.

Lemma 16.1. *Suppose that $z' = c(\bar{b}(t) - z)^3\phi(t) + \epsilon(t)$. If*

$$\|\bar{b}(t) - b\|_\infty \le \rho, \quad |E(t)| \le \delta, \quad c \ge (2\gamma^2 \textstyle\int_0^{1/2} \phi)^{-1},$$

we can define

$$z_\pm' = -c(z - b \mp \rho)^3\phi(t) \pm \delta$$

and look at its solution. Then $z_-(t) \le z(t) \le z_+(t)$ for all $t \in \mathbb{R}$.

So if we have this, we can prove something like

$$|b - z_\pm(\tfrac{1}{2})| < \gamma + \rho + \tfrac{1}{2}\delta.$$

To get an analytic ordinary differential equation simulating ψ, we use the functions

$$s(t) = \frac{1}{2}(\sin^2(2\pi t) + \sin(2\pi t)), \quad l_2(x, y) = \frac{1}{\pi}\arctan(4y(x - \tfrac{1}{2})) + \frac{1}{2}.$$

The reason we are using them is because they can be written as polynomial partial differential equations.

Definition 16.2. A function $f : \mathbb{R} \to \mathbb{R}$ is called a **polynomial initial value problem** if there is a polynomial ordinary differential equation such that

$$\vec{y}' = p(t, \vec{y}), \quad y(0) = y_0 \in \mathbb{R}^n,$$

such that the first component of the solution is f.

All these functions we wrote above are PIVPs, for instance, $f(t) = \sin t$ can be solved by $y_1' = y_2$ and $y_2' = -y_1$. Now we can write down

$$\begin{aligned} z_1' &= \lambda_1(f(r(z_2)) - z_1)^3\phi_1(t, z_1, z_2) + \epsilon_1(t), \\ z_2' &= \lambda_2(r(z_1) - z_2)^3\phi_2(t, z_1, z_2) + \epsilon_2(t), \\ \phi_1 &= l_2(r(-t), \tfrac{\lambda_1}{\gamma}(z_1 - f(r(z_2))))^4 + \tfrac{\lambda_1}{r} + 10), \\ \phi_2 &= l_2(r(t), \tfrac{\lambda_2}{\gamma}(z_2 - r(z_1)))^4 + \tfrac{\lambda_2}{\gamma} + 10). \end{aligned}$$

Let me just quickly sketch why this does the job. In the interval $[0, \frac{1}{2}]$, we have $|s(-t)| \leq \frac{1}{8}$ and so

$$\ell_2(s(-t), \tfrac{\lambda_1}{\gamma} \cdots) \leq \frac{1}{\frac{\lambda_1}{\gamma} \cdots}.$$

Then we see that

$$\phi_2 \leq \frac{\gamma}{\lambda_2(\|z_2 - r(z_1)\|^4 + 1)} \leq \frac{\gamma}{\lambda_2}\|z_2 - r(z_1)\|^{-3}.$$

This implies that

$$\|z_2'\|_\infty \leq \lambda_2\|r(z_1) - z_2\|^3\frac{\gamma}{\lambda_2}\|r - z_2\|^{-3} \leq \gamma.$$

By assumption, we have that $\|z_2(0) - x_0\|_\infty \leq \epsilon$, and so we have something like

$$\|z_2(t) - x_0\|_\infty \leq \|z_2(t) - z_2(0)\| + \|z_2(0) - x_0\| \leq \tfrac{1}{2}\gamma + \tfrac{1}{2}\delta + \epsilon.$$

This means that z_2 is almost kept constant. If we now look at the equation for z_1', we see that this almost looks like the lemma we had about perturbed differential equations. So we get

$$\|z_1(\tfrac{1}{2}) - \psi(x_0)\| < \epsilon.$$

Theorem 16.3. *If $x' = f(t, x)$ for $x(0) = x_0$ is a initial value problem, and $f : \mathbb{R}^{n+1} \to \mathbb{R}^n$ is a composition of polynomial initial value problems functions, then there exists a polynomial $p : \mathbb{R}^{n+1} \to \mathbb{R}^m$ and $y_0 \in \mathbb{R}^m$ such that the solution $(y_1(t), \dots, y_m(t))$ to*

$$y' = p(t, y), \quad y(0) = y_0 \in \mathbb{R}^m$$

satisfies $(y_1, \dots, y_n) = (x_1, \dots, x_n)$.

This is quite complicated to prove, but here is a demonstration. If we want to solve

$$x_1 = \sin^2 x_2, \quad x_2' = e^{x_1},$$

we can write $x_1 = (\sin x_2)^2$ and write $x_3 = \sin x_2$, so $x_3' = (\cos x_2)x_2'$, so you introduce new variables $x_4 = \cos x_2$. If you do this enough, you will see that

$$x_1' = x_3^2, \quad x_2' = x_5, \quad x_3' = x_4 x_5, \quad x_4' = -x_3 x_5, \quad x_5' = x_5 x_3^2.$$

This shows that Turing machines can be simulated by analogue machines.

17 November 12, 2018

Today we will talk about arithmetic circuits, due to Valiant.

17.1 Arithmetic circuits

This was an approach to prove $\mathcal{P} \neq \mathcal{NP}$, but this approach actually fails.

Definition 17.1. An **arithmetic circuit** consists of

- a variable set $X = \{x_1, \dots, x_n\}$,
- an operator set $O = \{+, \times\}$,
- a finite directed acyclic graph G with two kinds of vertices, input vertices V_0 of in-degree 0, output vertices V_2 with in-degree 2, and a unique output gate with out-degree 0,
- a labeling $V_0 \to \mathbb{C} \cup X$ and $V_2 \to O$.

The size of the circuit is the number of edges.

Definition 17.2. For a sequence $(f_n) \in \mathbb{C}[x_1, \dots, x_{m(n)}]_{d(n)}$ of polynomials, with d a polynomial in n, we say that (f_n) is in $\mathcal{VP}$ if there is a circuit (C_n) such that C_n computes f_n and the size of C_n is polynomial in n.

For example, Gaussian elimination suggests that $(\det_n(x))$ is in $\mathcal{VP}$. By $\det_n(x)$, we really mean the determinant of the $n \times n$ matrix (x_{ij}), as a polynomial in the variables x_{ij}. We can't really use Gaussian elimination here, because we aren't allowed to switch things.

Here is how you compute the determinant. We know that A^k can be computed effectively, so $\operatorname{tr}(A^k)$ can be computed effectively as well. Here, note that $\operatorname{tr}(A^k)$ are the sum of the eigenvalues $\sum \lambda_i^k$. To write the determinant in terms of these, we define

$$e_l(\lambda_1, \dots, \lambda_n) = \sum_{J \subseteq \{1,\dots,n\}, |J| = l} \lambda_{j_1} \cdots \lambda_{j_l}, \quad e_n = \det A.$$

Here, we can show that

$$e_l = \frac{1}{l!} \det \begin{pmatrix} p_1 & 1 & 0 & 0 & \cdots \\ p_2 & p_1 & 2 & 0 & \cdots \\ \vdots & p_2 & p_1 & 3 & \cdots \\ \vdots & \vdots & \ddots & \ddots & l-2 \\ p_{l-1} & \cdots & \cdots & p_2 & p_1 \end{pmatrix}.$$

Then the determinant of this nearly-lower-triangular matrix can be computed in $O(l^3)$.

Definition 17.3. We say that (f_n) is in $\mathcal{VNP}$ if there exists a sequence

$$g_n(x,y) \in \mathbb{C}[x_1,\dots,x_n,y_1,\dots,y_{m(n)}] \in \mathcal{VP}$$

such that

$$f_n(x) = \sum_{y\in\{0,1\}^{m(n)}} g_n(x,y).$$

Here is an example. The permanent

$$\mathrm{perm}_n(x) = \sum_{\sigma\in S_n} x_{1\sigma(1)}\cdots x_{n\sigma(n)} \in \mathcal{VNP}$$

is in $\mathcal{VNP}$. This is because we can define

$$g_n(x,y) = u_n(x,y)v_n(x,y),$$

$$u_n(x,y) = \prod_{(i=j)\leftrightarrow(l\neq m)} (1-y_{il}y_{jm}) \prod_i \biggl(\sum_j y_{ij}\biggr), \quad v_n(x,y) = \prod_i \sum_j x_{ij}y_{ij}.$$

Here, $u_n(x,y)$ is just picking up 1 exactly when y_{ij} form a permutation matrix. Then $v_n(x,y)$ computes the other thing.

Theorem 17.4. *The permanent* (perm_n) *is universal.*

17.2 Universality

This means that you get any polynomial as an affine projection of the permanent matrix.

Definition 17.5. We say that $p(x) \in \mathbb{C}[x_1,\dots,x_n]$ is an **affine linear projection** of $q(y) \in \mathbb{C}[y_1,\dots,y_m]$ if there is an affine linear map $\mathbb{C}^n \to \mathbb{C}^m$ such that such that $p(x) = q(y(x))$.

Definition 17.6. The **determinantal complexity** $\mathrm{dc}(p)$ of $p(x)$ is the smallest n such that p is an affine linear projection of $\det_n(y)$.

Here are some facts:

- For any polynomial p, we have $\mathrm{dc}(p) < \infty$. (This is the universality of the determinant.)
- For any polynomial p, $\mathrm{dc}(p)$ is at least the circuit complexity of p.
- We have $2^m - 1 \geq \mathrm{dc}(\mathrm{perm}_m) \geq \frac{m^2}{2}$.

Here is another way to formulate this. Consider an acyclic graph G (with unique source and target) with each edge labeled by a variable or a constant. For each path π, define

$$\mathrm{wt}(\pi) = \prod_{e\in\pi} \mathrm{wt}(e),$$

and then define

$$\mathrm{val}(G) = \sum_{\pi:s\to t} \mathrm{wt}(\pi).$$

Lemma 17.7. *If f is a formula (circuit with underlying graph being a tree) of size e, then there exists a graph G with*

$$|V(G)| \leq e+3, \quad |E(G)| \leq e+1$$

such that $\mathrm{val}(G) = f$.

Proof. You can induct on e. □

Let us now prove universality of the permanent. Given a graph G, you can define a new graph $\Gamma(G)$, given by

$$V(\Gamma(G)) = V(G)/(s \sim t), \quad E(\Gamma(G)) = \overline{E(G)} \cup \{\ell_\nu : \nu \neq [s]\},$$

where ℓ_ν is a loop at the vertex ν. Now choose an ordering of $V(\Gamma(G))$ and then write down the adjacency matrix $M(G)$ with weight

$$(M(G))_{ij} = \begin{cases} w(i \to j) & \text{if } i \to j \text{ is the image of an edge from } G \\ 1 & \text{if } (i \to j = i) = \ell_i, \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 17.8. *We have* $\mathrm{perm}(M(G)) = \mathrm{val}(G)$.

Proof. Look at the cycle decomposition of the elements of S_n and see which ones can have nonzero contribution. □

Theorem 17.9. (perm_n) *is $\mathcal{VNP}$-hard.*

This follows from the following proposition.

Proposition 17.10. *Let $g \in \mathbb{C}[x,y]$. If g has a formula of size e, then $f(x) = \sum_y g(x,y)$ can be realized as an affine linear projection of* perm_{6e}.

So the strategy is, start with an efficient graph that computes G. Add some components to G to construct a graph G' such that $\mathrm{perm}(M(G')) = f$.

Index