COMPUTABILITY AND COMPLEXITY 24 SPACE COMPLEXITY

AMIR YEHUDAYOFF
DIKU

Remark. *Computational complexity studies the resources that are needed to achieve computational tasks. On a high-level, computational devices have costs (like time, memory size, energy, randomness, training data, etc.), and computational tasks have complexities (the minimum cost that is needed to achieve it). We now move to focussing on space.*

1. Space

Example 1. *What is $\sqrt{2}$? Can we write its digits? What is π?*

Example 2. *Sometimes programs run out of memory.*

Example 3. *What is the space complexity of "palinodromes"?*

We will focus on space in the TM model.

Definition 4. *The space that a TM M uses on $x \in \{0,1\}^*$ is the number of locations on the (working) tape M visits during the run on x. It is denote by $\mathsf{SPACE}(M,x)$, and it could be infinite. For $n \in \mathbb{N}$, define*

$$\mathsf{SPACE}(M,n) = \max_{x \in \{0,1\}^n} \mathsf{SPACE}(M,x).$$

Remark. *Memory locations can be used more than once for free. Only the first time a position on the tape is visited is counted. This is what distinguishes space from time.*

Remark. *We do not count the access to the input tape because we want to consider algorithms with sublinear space.*

Definition 5. *Let $S : \mathbb{N} \to \mathbb{N}$. We say that $L \subseteq \{0,1\}^*$ is in $\mathsf{SPACE}(S(n))$ if there is a TM M that decides L so that*

$$\mathsf{SPACE}(M,n) = O(S(n)).$$

Recall that we introduced a powerful computational resources: non determinism ("the power to guess correctly" or $\exists$).

Definition 6. *The space cost of a NTM M on x is the maximum space in the run of M on x over all (non-net.) choices. We similarly get $\mathsf{NSPACE}(S(n))$.*

Remark. *We only deal with $S(n)$ that are "space-constructible". That is, there is a TM M that given 1^n as input, computes $S(n)$ using space $O(S(n))$. All functions $S(n)$ from now on are assumed to be such (without explicitly stating it). All reasonable functions are such (but not all functions are such). We shall ignore this issue from now on.*

How is space related to time?

Theorem 7. *For every $S : \mathbb{N} \to \mathbb{N}$ so that $S(n) \geq \log n$,*

$$\mathsf{DTIME}(S(n)) \subseteq \mathsf{SPACE}(S(n)) \subseteq \mathsf{NSPACE}(S(n)) \subseteq \mathsf{DTIME}(2^{O(S(n))}).$$

The first inclusion holds because in times T, a TM can visit at most T locations. The second holds because non-determinism just adds power. The third shall follow from the following discussion.

1.1. **Configuration graphs.**

We can represent computations by graphs.

Definition 8. *The configuration graph of M on x is a directed graph $G_{M,x}$ that is defined as follows. The vertices are the "configuration" of the run (i.e., a vertex v fully encodes a possible state of the computation). There is an edge (v, u) in the graph if v is a vertex, u is a vertex and M moves from state v to state u in one time step.*

Remark. *If M is deterministic, all out-degrees are one.*

Remark. *If M is non-deterministic, all out-degrees are at most two.*

Remark. *Loops in the graph correspond to computations that do not terminate.*

Claim 9. *M accepts x iff there is a path from the initial state to an accepting state in $G_{M,x}$.*

Claim 10. *Each vertex in the graph can be described using $O(\mathsf{SPACE}(M, x))$ bits.*

Claim 11. *The number of vertices in the graph is at most $2^{O(\mathsf{SPACE}(M,x))}$.*

As was explained in the proof of the Cook-Levin theorem, the edges can be described effectively:

Claim 12. *For every TM or NTM M and $x \in \{0, 1\}^n$, there is a CNF formula $\varphi_{M,x}$ so that if v, u are two bit strings (encoding potential vertices in the graph) then $\varphi_{M,x}(v, u) = 1$ iff both v, u are vertices and (v, u) is an edge in $G_{M,x}$. In addition, the size of $\varphi_{M,x}$ is at most $O(\mathsf{SPACE}(M, x) + \log n)$.*

Remark. *Intuitively, the formula $\varphi_{M,x}$ checks that v encodes a proper state, u encodes a proper state, and the transition $v \to u$ agrees with M. The formula has the claimed size because of the local behavior of TMs.*

Proof of last part of Theorem 7. Given a NTM M and input x, we need to decide in time $2^{O(\mathsf{SPACE}(M,x))}$ if M accepts x or not. In other words, we need to decide if there is a path from the initial state to an accepting state in $G_{M,x}$. This can be done, e.g., using BSF on the graph $G_{M,x}$. □

Remark. *This is a powerful idea. We can think of a computation as a walk in a graph that is implicitly descried by M and x. Reachability problem are thus deeply related to computation.*

Remark. *We see that basic algorithms, like BFS, help to understand things concerning general TMs.*

1.2. **Important classes.** There are a few natural choices for classes to focus on:

Definition 13.

$$\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{SPACE}(n^k)$$
$$\mathsf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)$$

Remark. $\mathsf{P} \subseteq \mathsf{PSPACE}$.

Remark. $\mathsf{NP} \subseteq \mathsf{PSPACE}$ *because e.g. we can check all assignments for a formula in polynomial space (and exponential time).*

Remark. *We do not know if* $\mathsf{P} = \mathsf{PSPACE}$ *or not. But if* $\mathsf{P} = \mathsf{PSPACE}$ *then* $\mathsf{P} = \mathsf{NP}$.

Remark. *We can represent 2^n objects in space n. So, allowing algorithms to run in space n may allow them to check 2^n options. This is often too costly. The following two classes are the space analogs of* P *and* NP.

Definition 14.

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$
$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

Remark (recap)**.** *Computations can be thought of as walks on huge digraphs. The edges of the graph are, however, easily described. A computational is "accepting" if two vertices are connected.*

1.3. PSPACE.

Remark. *One of the first things to do in order to understand a complexity class is find a "complete" problem for it.*

Remark. *A basic idea in this theory is that of "reductions". Intuitively, $A \leq B$ if "problem A is easier than B". We have seen $A \leq_p B$ where the reduction $f : \{0,1\}^* \to \{0,1\}^*$ so that $x \in A$ iff $f(x) \in B$ is poly-time. Later, we shall see other types of reductions. The reductions should fit the scenario we are thinking about.*

Definition 15. *A language $C \subseteq \{0,1\}^*$ is* PSPACE*-complete if $C \in \mathsf{PSPACE}$ and $L \leq_p C$ for every $L \in \mathsf{PSPACE}$.*

Remark. *There is a general way to construct complete problems. For* PSPACE*, it is something like*

$$\{\langle M, x, 1^s\rangle : M(x) = 1, \mathsf{SPACE}(M, x) \leq s\}.$$

But we typically want to identify more natural complete problems.

The complete problem we shall talk about is "totally quantified boolean formulas".

Definition 16. *A boolean formula over the variables $x_1, \ldots, x_n$ is an expression of the form*

$$(x_1 \wedge x_2) \vee (x_1 \vee (\neg x_2)).$$

It can be thought of as a tree (illustrate the example). Formulas can be inductively defined as follows. Each of the expression $x_1, \ldots, x_n$ and $0, 1$ are formulas. If f is *a formula then* $(\neg f)$ *is a formula. If* f_1, f_2 *are formulas then* $(f_1 \wedge f_2)$ *and* $(f_1 \vee f_2)$ *are formulas.*

Remark. *Formulas can be thought of as trees.*

Remark. *A formula computes a boolean function* $\{0,1\}^n \to \{0,1\}$ *in the obvious way.*

Remark. *Every boolean function* $\{0,1\}^n \to \{0,1\}$ *can be computed by some formula (in fact, by many formulas).*

Remark. *Formulas can be thought of as computational devices. As such, a formula has a cost. The size of* f *is inductively defined: the size of the base case is* 1*, and* $size(\neg f) = size(f)$ *and* $size(f_1 * f_2) = size(f_1) + size(f_2)$ *for* $* \in \{\wedge, \vee\}$.

Remark. *If* f *has size* s *then it can be described by* $O(s)$ *bits.*

Remark. *Formulas lead to a different model of computational complexity theory; more on this later on. In it, the devices are formulas and the complexity of a function is the size of the minimum formula computing it.*

Remark. *A totally quantified boolean formula is an expression of the form*

$$\forall x_1 \in \{0,1\} \exists x_2 \in \{0,1\}\ (x_1 \vee x_2) \wedge x_1.$$

The quantifiers Q *can be either* $\forall$ *and* $\exists$*. A totally quantified boolean formula (TQBF) is an expression of the form*

$$E = Q_1 x_1 Q_2 x_2 \dots Q_n x_n\ \varphi$$

where φ *is boolean formula over* $x_1, \dots, x_n$*. It is understood that each* x_i *takes values in* $\{0,1\}$*. Every TQBF has a truth value in* $\{0,1\}$*. For example, the truth value of the above expression is* 1.

Example 17.

$$\phi(x_1, \dots, x_n) \in \mathsf{SAT} \iff 1 = \exists x_1 \dots \exists x_n \phi$$

Remark. *TQBF capture properties that are deeply related to game theory. Let us consider chess for example. The expression*

$$\exists w_1 \forall b_1 \exists w_2 \forall b_2 \dots \exists w_{100}\ \textit{white wins}$$

says that there is a first move for white, so that for every move of black, there is a move of white, ... so that white wins in 100 *moves.*

Definition 18.

$$\mathsf{TQBF} = \{\langle E \rangle : E \textit{ is a true TQBF}\}.$$

Theorem 19. TQBF *is* PSPACE*-complete.*

Proof. We start by sketching why TQBF is in PSPACE. This is not entirely trivial because we need to reuse space (there are exponentially many things to check). The algorithm A is recursive. The base case is boolean formulas over $0, 1$. The algorithm A output the truth value of the input formula. The inductive step is as follows. If the input to A is

$$\forall x_1 \psi(x_1)$$

for some (partially quantified) formula ψ, the algorithms first substitute $x_1 = 0$ in ψ, runs the recursion and get $y_0 = A(\psi(0))$. It writes y_0 down and deletes the working memory. The algorithms then computes $y_1 = A(\psi(1))$ and outputs $y_0 \wedge y_1$. The case of $\exists x_1 \psi(x_1)$ is similar. The memory size is

$$O(\text{the number of quantifiers plus the size of the inner formula}).$$

It remains to prove that it is PSPACE-hard. Let $L \in \mathsf{PSPACE}$ and let M be a TM with poly-space deciding M and fix an input x. Let

$$s = \mathsf{SPACE}(M, x).$$

We wish to construct a TQBF ψ so that

$$M(x) = 1 \iff \psi = 1.$$

We translate $M(x) = 1$ to the reachability problem in $G_{M,x}$ from the initial state to an accepting state. We want ψ to capture this reachability.

> How can we do that? We want to capture "there is a path". We have $\exists$ at our disposal. But how can $\forall$ help?

We discussed a formula $\varphi = \varphi_{M,x}$ that computes if v, u is an edge in $G_{M,x}$. In other words, the formula

$$\varphi(v, u)$$

checks if there is a path of length one from v to u. The formula

$$\exists w \ \varphi(v, w) \wedge \varphi(w, u)$$

checks if there is a path of length at most two from v to u.

> Can we keep on going?

The answer is, on the face of it, no because the length of the desired path could be exponential in s and so will the size of the overall formula we get. And we also did not use the $\forall$ quantifier.

> The main observation is that if $\phi_i(v, u)$ computes if there is a path of length at most 2^i between v, u, then
>
> $$\phi_{i+1}(v, u) = \exists w \ \forall a, b \ ((a = v) \wedge (b = w)) \vee ((a = w) \wedge (b = u)) \to \phi_i(a, b)$$
>
> computes if there is a path of length at most 2^{i+1} between v, u.

In words, there is some w so that $\phi_i(v, w)$ and $\phi_i(w, u)$.

> We doubled the distance covered and increased the formula size by an additive factor.

Recall that v, u, w are $O(s)$-bit strings. So, the size of ϕ_{i+1} is at most $O(s)$ plus the size of ϕ_i. The final formula is chosen to be $\psi = \phi_{O(s)}$ and its size is $O(s^2)$. It holds that

$$\psi(v_0, v_{accept}) = 1 \iff M(x) = 1.$$

□

Remark. *The theorem can be thought of saying that TQBF is "extremely hard". In other words, in general deciding if a player in a two-player game (like chess) has a winning strategy is hard.*

Remark. *What properties of $G = G_{M,x}$ did the proof use? It has size $2^{poly(n)}$. Deciding the edges of G can be done in time $poly(n)$ and space $poly(n)$. We never used the fact that the out-degrees are one. So it applies both to deterministic as well as non-deterministic computations!*

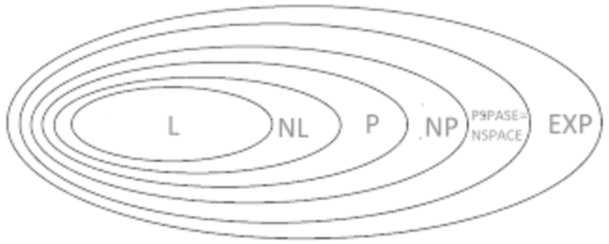
Theorem 20. $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

The ideas we have developed, in fact, allow to prove the following general result:

Theorem 21 (Savitch 1970)**.** *If $S(n) \geq \log n$ then*

$$\mathsf{NSPACE}(S(n)) \subseteq \mathsf{SPACE}((S(n))^2).$$

Remark. *A standard picture in computational complexity looks like:*



1.4. **NL.**

Remark. *Again, we try to identify complete problems for* NL. *What is the suitable notion of reductions? Allowing reductions to run in poly-time is not suitable because* $\mathsf{NL} \subseteq \mathsf{P}$.

Definition 22. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is implicitly log-space computable (ILC) if*

(1) *there is $k > 0$ so that for all x,*

$$|f(x)| \leq |x|^k.$$

(2)

$$\{\langle x, i\rangle : f(x)_i = 1\} \in \mathsf{L}.$$

(3)

$$\{\langle x, i\rangle : i \leq |f(x)|\} \in \mathsf{L}.$$

Remark. *The first item says that f does not output very long strings. The other two items say that we can compute in log-space the i'th bit of $f(x)$ as long as it makes sense.*

Definition 23. *We write $A \leq_\ell B$ if there is an ILC f so that for all x, we have $x \in A$ iff $f(x) \in B$.*

Remark. *This type of reductions satisfy the following two natural properties:*

(1) *If $A \leq_\ell B$ and $B \in \mathsf{L}$ then*

$$A \in \mathsf{L}.$$

(2) *If $A \leq_\ell B$ and $B \leq_\ell C$ then*

$$A \leq_\ell C.$$

This is intuitive but not obvious; space must be reused here. The details are left as an exercise.

Definition 24. *A language $C \subseteq \{0,1\}^*$ is* NL*-complete if it is in* NL *and for every $L \in \mathsf{NL}$ we have $L \leq_\ell C$.*

Theorem 25. *The language*

$$\mathsf{PATH} = \{\langle G, s, t\rangle : G \textit{ is a digraph, } s, t \in V(G)\textit{, there is a path from } s \textit{ to } t\}$$

is NL*-complete.*

Remark. *We shall not fully prove but we have seen all ideas. First,* $\mathsf{PATH} \in \mathsf{NL}$ *because we can guess the path from* s *and accept only if we reach* t*. In this process, we "forget the history". Second, if* $L \in \mathsf{NL}$ *then there is a NTM* N *that decides it that uses log-space, so we can define a reduction:*

$$f(x) = \langle G_{N,x}, v_0, v_{accept} \rangle.$$

2. Randomness

Remark. *We now move to discuss a different computational resource: randomness.*

Remark. *What is "randomness"? We shall use the language of mathematics. In mathematics, randomness corresponds to a probability space. We shall work only with finite spaces.*

Remark. *What is randomness good for? It can help to hide things. It can help in algorithm-design. It can help to avoid "worst-case" choices.*

Remark. *How do we generate randomness? We can toss coins; this requires some device or person. We can use some internal "noise" in computers. We can use physical phenomena, like radioactive decay. We are not going to discuss any of these "engineering" problems.*

2.1. **The basics.**

Remark. *We shall introduce randomness into the Turing machine model as follows. A probabilistic Turing machine (PTM) M has three tapes: input tape, working tape, and randomness tape. There are two types of inputs to the machine. The usual x and a random string R. The string R will consist of i.i.d. uniform bits (but this could be chanced according to context).*

Definition 26. *For $T : \mathbb{N} \to \mathbb{N}$, a language $L \subseteq \{0,1\}^*$ is in $\mathsf{BPTIME}(T(n))$ if there is a PTM M so that for all $x \in \{0,1\}^n$,*

$$\Pr[M(x,R) = 1] \geq \frac{2}{3} \iff x \in L$$

where

- *R is uniformly distributed in $\{0,1\}^{O(T(n))}$.*
- *$\mathsf{TIME}(M,x,R) \leq O(T(n))$ for all R.*

Remark. *As we shall see later on, the $\frac{2}{3}$ is not important.*

Definition 27. *The class of bounded-error probabilistic polynomial time language is*

$$\mathsf{BPP} = \bigcup_{k \in \mathbb{N}} \mathsf{BPTIME}(n^k).$$

Remark.

$$\mathsf{P} \subseteq \mathsf{BPP}.$$

The question

is $\mathsf{P} = \mathsf{BPP}$?

is central and open. It asks whether we can always efficiently de-randomize computations.

2.2. **An example: polynomial identity testing (PIT).**

Remark. *If $A, B \in \{0,1\}^n$, how much time does it take to check if $A = B$ or not? Can randomness help to reduce the running time?*

Remark. *Polynomials are an example where randomness can help. This property is very important; for example, for error correction.*

Remark. *We work with n variables $x_1, \dots, x_n$ over the field of rational numbers (it could also be other fields). A monomial is an expression of the form*

$$x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$$

where e_i is a non-negative integer and by convention $x^0 = 1$. The degree of this monomial is $\sum_{i \in [n]} e_i$. We shall denote this monomial by x^e where $e = (e_1, \dots, e_n)$. A polynomial is a (finite) linear combination of monomials

$$p(x) = \sum_e \alpha_e x^e$$

where $\alpha_e \in \mathbb{Q}$. That number α_e is called the coefficient of x^e in p. The degree of p is the maximum degree of a monomial that appears in p.

Example 28. *The degree of $x_1 + x_1 x_2 - x_1 x_2^3$ is four.*

We would like to check if two polynomials p, q are equal or not. Over the rationals, there are two ways to think about this equality. First, as formal sums. Second, as functions $\mathbb{Q}^n \to \mathbb{Q}$. (Over other field, the two notions may not be the same.)

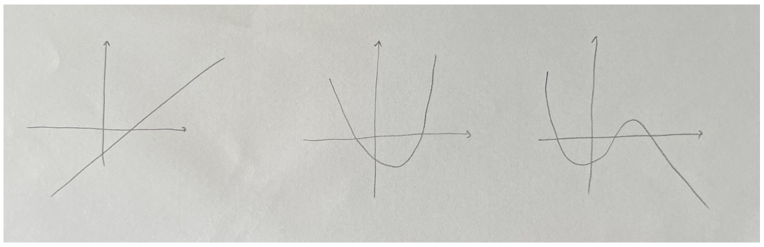
Remark. *Let's say we have black-box access to p, q. That is, we can choose $a \in \mathbb{Q}^n$ as we wish and ask "is $p(a) = q(a)$?"*

How many questions of this form we need to perform?

The answer depends on the degrees. If both degrees are at most d, then naively we need something like d^n questions, which is a lot. Randomness allows to reduce the number of questions! This relies on the fact that polynomials have few roots.

Lemma 29. *A univariate polynomial p of degree d has at most d roots.*

Remark. *We shall not prove (do you know how to prove?).*



Lemma 30 (DeMillo–Lipton–Schwartz–Zippel)**.** *If $p(x)$ is a non-zero polynomial in n variables of degree d and $S \subseteq \mathbb{Q}$ is non-empty, then*

$$\Pr[p(R) = 0] \leq \frac{d}{|S|}$$

where $R = (R_1, \dots, R_n)$ is uniformly distributed in S^n.

Remark. *The lemma shows that with just one random question (from a large collection of questions) we can check if $p = q$ or not.*

Proof. The proof is by induction on n. The base case $n = 1$ holds because univariate polynomial of degree d has at most d roots. The induction step is performed as follows. Write

$$p(x) = \sum_{i=0}^{d} p_i(x_1, \dots, x_{n-1}) x_n^i$$

where each p_i is in $n-1$ variables of degree at most $d-i$. Because p is non-zero, there is some i so that p_i is non-zero. Let i_* be the maximum i so that p_i is non-zero. By induction,

$$\Pr[p_{i_*}(R')=0] \le \frac{d-i}{|S|}$$

where $R'=(R_1,\ldots,R_{n-1})$. By the $n=1$ case,

$$\Pr[p(R)=0|R',p_{i_*}(R')\le 0] \le \frac{i_*}{|S|}.$$

Overall,

$$\begin{aligned}\Pr[p(R)=0] &= \Pr[p(R)=0,p_{i_*}(R')=0]+\Pr[p(R)=0,p_{i_*}(R')\neq 0]\\ &\le \Pr[p_{i_*}(R)=0]+\Pr[p(R)=0|p_{i_*}(R')\neq 0]\\ &\le \frac{d-i_*}{|S|}+\frac{i_*}{|S|}=\frac{d}{|S|}.\end{aligned}$$ □

2.3. **Representing polynomials.**

Remark. *Polynomials are extremely useful in many areas (mathematics, CS, science,...).*

Example 31. *Two central examples are over the $n\times n$ matrix of variables $X=(x_{i,j})$:*

$$\mathsf{det}_n(X)=\sum_{\pi\in S_n}\mathsf{sign}(\pi)\prod_{i=1}^n x_{i,\pi(i)}$$

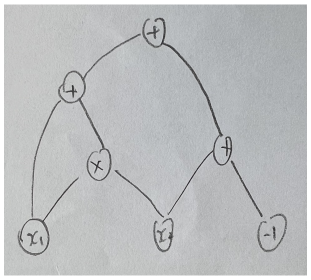
and

$$\mathsf{perm}_n(X)=\sum_{\pi\in S_n}\prod_{i=1}^n x_{i,\pi(i)}$$

where S_n is the (group of) permutations on $[n]$. The determinant is important in linear algebra and geometry (so in graphics, signal analysis, etc.). The permanent is important—it is the ultimate counting problem.

Remark. *We would like to represent polynomials efficiently. This leads to a new type of complexity theory: algebraic complexity.*

Definition 32. *An algebraic circuit is a DAG with in-degrees either zero or two. In-degree zero nodes are labelled by variables or field elements. In-degree two nodes are labelled by $+$ or $\times$. Circuits compute polynomials in the obvious way. The size of a circuit is the number of nodes in it.*



Definition 33. *The circuit complexity of p is the minimum size of a circuit computing p.*

Remark. *Again, devices have costs and tasks have complexities.*

Remark. *The circuit size of the* det_n *is at most* $O(n^3)$ *using Gaussian elimination. It is in fact at most* $O(n^{2.5})$ *which is much harder to see. The exact exponent is not known and the problem is important. (There is a long and interesting discussion here...)*

Remark. *The circuit complexity of* perm_n *is not known and it is one of the most important problems in theory of CS. But it is known that if there is an efficient way to compute* perm_n *then* $\mathsf{P} = \mathsf{NP}$ *and much more. (There is a long and interesting discussion here...)*

Remark. *Circuits have a different cost which captures the ability to perform the computation in parallel. The depth of the circuit is the length of the longest (directed) path in the graph.*

Remark. *Algebraic circuits have a very nice property: they can be balanced. If an algebraic circuit of size s computes a polynomial of degree d then there is circuit of size* $\mathsf{poly}(n, s, d)$ *and small depth* $O(\log^2(sd))$. *A similar result is not believed to hold for Boolean circuits.*

Example 34. *Here is a nice example of how PIT is related to other algorithmic problems, and how randomness could help. Let G be a bipartite graph where each color class is of size n. The problem we want to solve is*

"does G contain a perfect matching?"

There is a polynomial time algorithm for doing so. But it is not clear how to use randomness, or how can randomness be helpful for this problem. It turns out randomness is useful in allowing to solve the problem in "short parallel time with few processors".

Given an input graph G, define the $n \times n$ matrix of variables X:

$$X_{i,j} = \begin{cases} x_{i,j} & (i,j) \in E(G) \\ 0 & (i,j) \notin E(G) \end{cases}$$

Denote by $p(X)$ the determinant of X. The polynomial p has degree n in at most n^2 variables. A basic observation is that

$$G \text{ has a perfect matching (PM)} \iff \det(X) \neq 0.$$

If we set $S = [3n]$, and choose each $R_{i,j}$ i.i.d. uniform in S then

$$\Pr[p(R) = 0] \leq \frac{1}{3}.$$

In other words, the algorithm

if $p(R) = 0$ output "no PM" and if $p(R) \neq 0$ output "yes PM"

succeeds with probability at least $\frac{2}{3}$. As described above, $p(R)$ can be computed in $\mathsf{poly}(n)$*-time and* $O(\log^2(n))$*-depth. This is an efficient algorithms that can be run in short parallel time.*

2.4. **Back to PIT.**

Remark. *Instead of working with polynomials as black-boxes, we wish to work with explicit representation of them as algebraic circuits.*

Given two algebraic circuits C, C' over the integers $\mathbb{Z}$, we want to check if $C = C'$ in the sense that the polynomials they compute are the same.

Remark. *We think of the input length as the size s of the circuits.*

Claim 35. *If C has size s then its output has degree at most 2^s.*

Remark. *Using the previous approach (based on the DSZL lemma) would require computations with 2^s bits, which is too expensive. Again, randomness comes to the rescue. The idea is to perform the computation modulo some random large prime k.*

The algorithm.

(1) Let S be the set of integers between 0 and $10 \cdot 2^s$.
(2) Choose $R = (R_1, \ldots, R_n)$ uniformly in S^n.
(3) Choose a prime $k \leq N := 2^{4s}$ uniformly at random.
(4) If $C(R) \mod k = C'(R) \mod k$, output "equal" and otherwise output "not equal".

Running time. The running time is now polynomial in s, because all computations are modulo k.

Equality case. If $C = C'$ the algorithm *always* outputs the correct answer.

Inequality case. If $C \neq C'$ then analyze the chance of failure as follows. Let

$$y = C(R) - C(R').$$

We know that

$$\Pr[y = 0] \leq \frac{2^s}{|S|} = \frac{1}{10}.$$

The integer y is at most $(10 \cdot 2^s)^{2^s}$ and so when we decompose it to a product of primes, there are at most $\log((10 \cdot 2^s)^{2^s}) \leq 2^{3s}$ prime factors. The number of primes at most N is at least

$$\Omega\Big(\frac{N}{\log N}\Big) = \Big(\frac{2^{4s}}{s}\Big).$$

(This was conjectured by Gauss and by Legendre, and proved by Hadamard and by de la Vallee Poussin). So,

$$\Pr[y \mod k = 0] \leq O\Big(\frac{2^{3s}}{2^{4s}/s}\Big) \leq \frac{1}{10}$$

for large s. Overall, by the union bound,

$$\Pr[\text{error}] \leq \frac{2}{10}. \quad \square$$

Remark. *It is not trivial to sample a random prime efficiently, but it can be done.*

Remark. *We see how ideas from various areas of mathematics are deeply related to algorithm design.*

2.5. **RP.** The PIT algorithm we say has "one sided" error. Namely, if $C = C'$ the output is always correct and if $C \neq C'$ the output is correct with probability at least $\frac{2}{3}$. This is a stronger guarantee than BPP requires. This type of guarantee is also natural and important, and there is a special term for it. It is called "randomized polynomial time" and denoted by RP.

Remark. *There are other natural classes for randomized computations but we shall not go over all of them in this course.*

2.6. **Error reductions.** Given a PTM M for a language L and input x, we can think of $M(x)$ as random variable that is typically equal to $L(x)$:

$$\Pr[M(x) = L(x)] \geq \frac{2}{3}.$$

How can we increase the chance of success?

Run M a few times.

If $y_1, \ldots, y_T$ is the outcomes of the T independent runs of M on x then $\mathsf{MAJ}(y_1, \ldots, y_T)$ is much more likely to be $L(x)$.

Theorem 36 (concentration of measure)**.** *If $Z_1, \ldots, Z_n$ are i.i.d. variables taking values in $[0,1]$ with expectation $\mu = \mathbb{E}Z_1$, then for all $t \geq 0$,*

$$\Pr\left[\left|\Big(\frac{1}{n}\sum_{i\in[n]} Z_i\Big) - \mu\right| > t\right] \leq 2e^{-2t^2 n}.$$

Remark. *There are many inequalities of this from (by Chernoff, Hoeffding, Azuma, Bernstein, and more). They are important in many areas.*

Remark. *This shows that in BPP the failure probability can be between $2^{-\mathsf{poly}(n)}$ and $\frac{1}{2} - \frac{1}{\mathsf{poly}(n)}$ and the meaning of BPP will stay the same.*

Remark. *There are several way to prove concentration bounds. Some use the "moment method" or "Fourier transform". Others are more "algorithmic".*

2.7. **Two de-randomization conclusions.**

Theorem 37 (Adleman)**.** $\mathsf{BPP} \subseteq \mathsf{P/poly}$. *In other words, for every $L \in \mathsf{BPP}$ and $n \in \mathbb{N}$, there is a Boolean circuit computing*

$$\{0,1\}^n \ni x \mapsto L(x) \in \{0,1\}.$$

Sketch. The idea is that

(1) (as we said) we can assume that the error is smaller than 2^{-n}, and
(2) there are only 2^n inputs.

So, we can apply the union bound. (The details are left as an exercise.) □

Theorem 38 (Sipser-Gacs)**.**

$$\mathsf{BPP} \subseteq \Sigma_2.$$

Because BPP *is closed under complements, we can deduce*

$$\mathsf{BPP} \subseteq \Sigma_2 \cap \Pi_2.$$

Sketch. We have a PTM $P(x, R)$ that decides $x \in L$. The randomness R is in $\{0,1\}^m$ for $m = \mathsf{poly}(n)$. In fact, we can assume

$$x \in L \Rightarrow \Pr[M(x, R) = 1] \geq \frac{1}{2}$$

and

$$x \notin L \Rightarrow \Pr[M(x, R) = 1] < \frac{1}{2m+1}.$$

(See that you understand the latter condition when $x \notin L$.) We should construct a poly-time TM M so that for all inputs x,

$$x \in L \iff \exists a \forall b M(x, a, b) = 1.$$

What are a and b?

Fix $x \in \{0,1\}^n$. Consider the subset of strings that lead M to accept

$$S_x = \{R : M(x, R) = 1\}.$$

The size of S_x tells us if $x \in L$ or not. If $|S_x|$ is large, we should accept, and otherwise we should reject.

How can we do that?

Here is the main idea. For every $u \in \{0,1\}^m$, let

$$u + S_x = \{u + s : s \in S_x\}$$

where addition is in $\{0,1\}^m$ (entry-wise modulo two).

Claim 39. *If $|S_x| \geq \frac{1}{2} \cdot 2^m$ then there are $t \leq m+1$ vectors $u_1, \ldots, u_t$ so that*

$$\bigcup_{i \in [t]} (u_i + S_x) = \{0,1\}^m.$$

Remark. *If u is uniformly at random, then $u + S_x$ is a random subset of fractional size $\geq \frac{1}{2}$. It is not distributed uniformly at random, but each element is in it with probability at least a half.*

Proof idea. Taking $O(\log 2^m) = O(m)$ sets allows to cover the whole universe. With one set, we cover $\frac{1}{2}$. With two sets, we cover $1 - \frac{1}{4}$. With threes sets, we cover $1 - \frac{1}{8}$. And so forth, until we cover more than $1 - 2^m$ of the domain (at which point we are done). $\square$

The witness a is now

$$a = (u_1, \ldots, u_t).$$

And b allows to check if indeed

$$\bigcup_{i \in [t]} (u_i + S_x) = \{0,1\}^m.$$

Namely, b is an element in $\{0,1\}^m$ and

$$M(x, a, b) = 1\big[\exists i \in [t] \ P(x, b + u_i) = 1\big].$$

We see that

$$x \in L \Rightarrow \exists a \forall b \ M(x, a, b) = 1.$$

What about the "no case"?

When $x \notin L$, as we already justified:

$$|S_x| < \frac{1}{t}2^m$$

so that for all $u_1, \ldots, u_t$,

$$\Big|\bigcup_{i\in[t]}(u + S_x)\Big| < t \cdot |S_x| < 2^m;$$

here we used the property of M for $x \notin L$. Stated differently,

$$x \notin L \Rightarrow \forall a \exists b \; M(x, a, b) = 0.$$

□