Computational Complexity using Deterministic, Randomized, and Quantum Computers

G. Eric Moorhouse, UW Math

References

H.-K. Lo, S. Popescu, and T. Spiller, *Introduction to Quantum Computation and Information,* 1998.

C.P. Williams and S.H. Clearwater, *Explorations in Quantum Computing,* 1998.

A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* 1974.

P. Shor, 'Quantum computing', proceedings of the International Congress of Mathematicians, 1998.
`http://www.research.att.com/~shor/papers/ICM.pdf`

P. Shor, 'Polynomial-time algorithms for prime factorization and discrete logarithm problems', *SIAM J. Computing* **26** (1997), 1484-1509.
`http://www.research.att.com/~shor/papers/QCjournal.pdf`

Models of Computers and Computability

We will describe

(i) RAM's (Random Access Machines);

(ii) Turing machines;

(iii) logical circuits; and

(iv) non-deterministic, randomized and quantum versions of (i)–(iii).

Two types of computational problems:

- decision problems (yes/no answers only);
- arbitrary computational problems.

RAM's (Random Access Machines)

Infinitely many memory locations

$$M_0, M_1, M_2, \ldots,$$

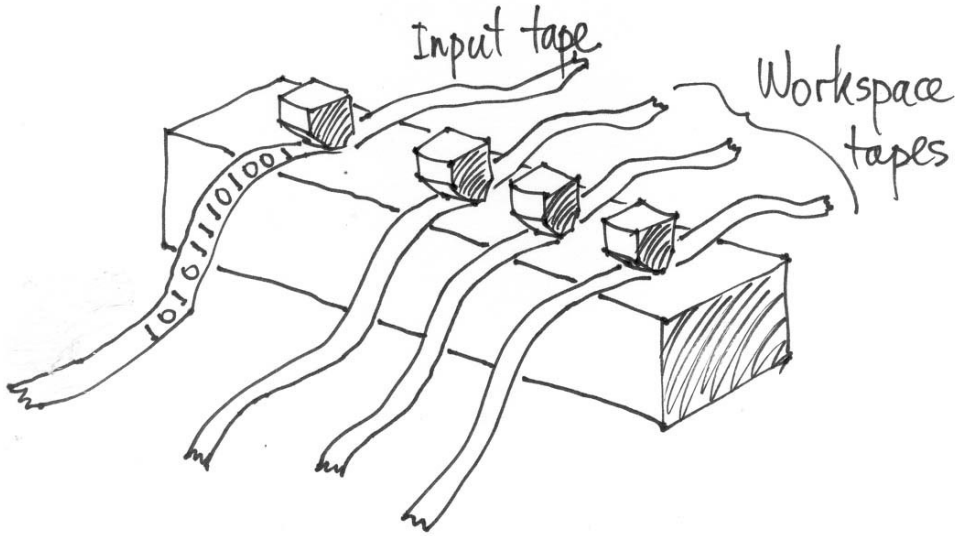
each of which can hold any integer. Each location can be accessed immediately, and we can use some locations as pointers to other locations.

Uses an idealized, simplified programming language much like the assembly language of any modern computer.

Turing Machines

Take

- an *alphabet* $\Gamma \supseteq \{0, 1, blank\}$;
- a finite set Q of states, including
  a designated *initial state* $q_0 \in Q$ and
  a subset of *accepting states* $A \subseteq Q$;
- an input tape and k workspace tapes, all of which are readable and writable; and
- a *transition function*
  $\delta : Q \times \Gamma^{k+1} \to Q \times \Gamma^{k+1} \times \{L, R\}^{k+1}$.



The machine runs by iterating δ until δ is undefined for the current state and symbols read, at which point the machine *halts.* An input (a string of symbols from the alphabet Γ fed in via the read/write tape) is *accepted* by the machine if it halts in an accepting state. The *language* of the machine is the set of words (input strings) accepted by the machine.

An arbitrary language L (set of words, i.e. strings from Γ) is *decidable* if there exists a Turing machine which eventually halts for every possible input, and whose language is L.

Sample Turing Machine

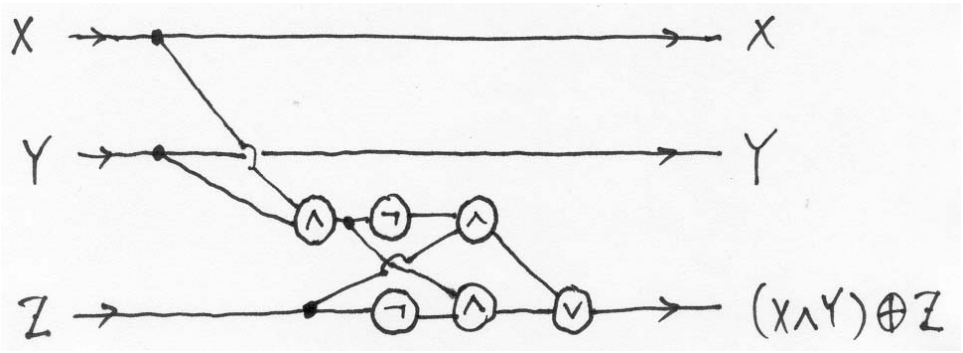
This machine accepts words in the language L of all finite strings consisting of the *same number* of 0's and 1's.

Transition function δ

INPUT			OUTPUT			
State	Input tape	Work tape	State	Work tape	Move input	Move work
q_0	0	blank	q_1	0	R	L
q_0	1	blank	q_1	0	R	R
q_1	0	blank	q_1	blank	R	L
q_1	0	0	q_1	0	R	L
q_1	1	blank	q_1	blank	R	R
q_1	1	0	q_1	0	R	R
q_1	blank	0	q_2			

Here q_0 is the initial state, q_2 is the accepting state. I use $k = 1$ workspace tape.

Logical Circuits



Example: Toffoli Gate

X	Y	Z	$(X \wedge Y) \oplus Z$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

This gate is *universal* for classical logic.

The Church–Turing Hypothesis ("Church's Thesis"):

Any 'effectively computable function' can be computed by a Turing machine.

Support for this hypothesis is given by

Theorem. *The following classes of problems are identical:*
(i) The class of computational problems solvable by a **Turing machine***.*
(ii) The class of computational problems solvable by a **RAM***.*
(iii) The class of computational problems solvable by a family of **logical circuits** *that can be generated by a Turing machine.*

Polynomial Time

$\mathcal{P} =$ the class of decision problems solvable in 'polynomial time' (i.e. time $O(\ell^k)$ for some $k \geq 0$, where ℓ is the input size) on
(i) a Turing machine; or
(ii) a RAM*; or
(iii) a family of circuits† of polynomial size.

These are roughly the 'manageable' problems.

$\mathcal{P} =$ the class of languages accepted by some (deterministic) Turing machine which halts in time polynomial in the input size.

---

*RAM log-cost model: the time required to add two ℓ-bit numbers is $O(\ell)$.

†We assume the circuit for inputs of size ℓ can be generated by a Turing machine with running time $O(\ell^k)$.

Non-deterministic Turing Machines

Replace the deterministic transition function δ by a relation which gives a *set* of possible transitions for each input. A word is *accepted* by the machine if, for this input, there *exists* a set of transitions allowed by δ for which the machine halts in an accepting state. The *language* of a machine is the set of all words accepted by a non-deterministic Turing machine which halts for all possible computational routes.

Equivalently, the machine is allowed to make 'guesses'. We can implement this machine using a deterministic Turing machine with an additional read-only tape for inputing guesses.

$\mathcal{NP}$ = the class of decision problems that can be solved in *polynomial* time on some *non-deterministic* Turing machine which halts for all possible 'guesses'

= the class of all languages accepted by some non-deterministic Turing machine with execution time $O(\ell^k)$ for some $k \geq 0$.

Consider a decision problem: test whether $x \in L$ for a given language L. Testing whether $x \in L$ in polynomial time on a non-deterministic Turing machine, is equivalent to checking a given proof that $x \in L$ in polynomial time on a deterministic Turing machine.

Example: Factoring n is in $\mathcal{NP}$

Given n and two guesses p,q for the factors of n, we can easily check in polynomial time whether or not $n = pq$. (Multiplication of ℓ-digit numbers p, q is performed asymptotically in time $O(\ell \log \ell \log\log \ell)$; or practically in time $O(\ell^2)$.)

Big Open Problem

Clearly $\mathcal{P} \subseteq \mathcal{NP}$. Does $\mathcal{P} = \mathcal{NP}$?

A decision problem $x \in L$ is $\mathcal{NP}$-*complete* if it is in $\mathcal{NP}$, and if every $\mathcal{NP}$ decision problem can be reduced (in polynomial time) to the problem $x \in L$. Thus $\mathcal{P} = \mathcal{NP}$ iff $L \in \mathcal{P}$.

Thousands of useful problems have been shown to be $\mathcal{NP}$-complete, e.g. the *satisfiability problem:* Given a logical expression similar to

$$(P_1 \vee P_4 \vee P_5) \wedge (P_2 \vee P_4 \vee P_9) \wedge \cdots \wedge (P_4 \vee P_{11} \vee P_{26}),$$

determine whether or not there is an assignment of values $\{T, F\}$ to $P_1, P_2, P_3, \ldots$ for which the expression is true (S.A. Cook, 1971).

Randomized Turing Machines

Like a (deterministic) Turing machine, but we allow the machine to flip coins during the computation, and transitions can depend on the outcomes of the flips.

Equivalently, a *randomized Turing machine* can be implemented as a deterministic Turing machine with an additional read-only input tape for inputing a random stream of 0's and 1's.

$\mathcal{BPP}$ = the class of languages L for which there exists a polynomial-time randomized Turing machine which accepts each word $x \in L$ with probability at least $\frac{2}{3}$, and rejects each $x \notin L$ with probability at least $\frac{2}{3}$.

E.g. the set of all primes (considered as a language) is in $\mathcal{BPP}$. Moreover if the Generalized Riemann Hypothesis holds, then the set of all primes is in $\mathcal{P}$ (Miller, 1976).

Benchmark Problems for Randomized Computation

Factorization: Given an integer N, compute the prime factorization of N.

The limit for factorization using current algorithms on modern computers is about 120 decimal digits. Expected execution time

$$O(e^{\ell^{1/3}(\log \ell)^{2/3}(C+o(1))})$$

where $\ell = \log(N)$ (the number of digits of N). See A.K. Lenstra and H.W. Lenstra, 1993.

Proving Primality: Given an integer N, which we suspect to be prime, prove that N is prime.

Current algorithms on modern computers are able to prove primality of numbers up to about 1000 decimal digits. Expected execution time

$$O(\ell^6).$$

See L. Adleman and M. Huang, 1992.

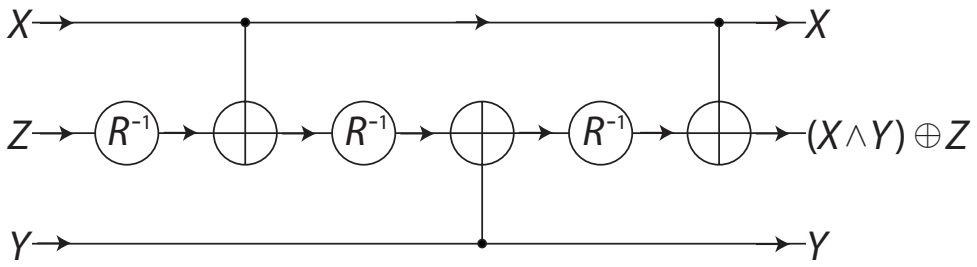
Proving Compositeness: Expected execution time $O(\ell^2)$ (Rabin 1980, Miller 1976).

Quantum Circuits

Similar to classical logical circuits. There are ℓ input 'wires' and ℓ output 'wires', each carrying (rather than a bit, i.e. binary value 0 or 1) a value $(z_0, z_1) \in \mathbb{C}^2$, called a *qubit.* In between, information moves from left to right, passing through certain quantum gates, each gate having either one input and one output wire, or two input and two output wires. At each stage of the computation, the *state* of the system is represented by a unit vector in $\bigotimes^n \mathbb{C}^2 \cong \mathbb{C}^{2^n}$. Each gate performs a unitary transformation A on the one or two qubits to which it is applied. The effect of each gate on the entire state of the system is obtained by tensoring A with the identity on the remaining qubits (to obtain an element of $U(2^n, \mathbb{C})$). The net effect of the entire circuit is the composition (i.e. matrix product) of the matrices in $U(2^n, \mathbb{C})$ corresponding to the individual gates.

(One must also impose a 'uniformity condition', similar to our restriction on classical logical circuits, to ensure that the complexity of the circuit itself is not more than polynomial in the input size ℓ.)

Example: A quantum circuit which simulates a classical Toffoli gate.



$$R = \begin{pmatrix} \cos(\frac{\pi}{8}) & \sin(\frac{\pi}{8}) \\ -\sin(\frac{\pi}{8}) & \cos(\frac{\pi}{8}) \end{pmatrix}, \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{bmatrix} R^{-1} & & & \\ & R^{-1} & & \\ & & R^{-1} & \\ & & & R^{-1} \end{bmatrix} \begin{bmatrix} I & & & \\ & W & & \\ & & I & \\ & & & W \end{bmatrix} \begin{bmatrix} R^{-1} & & & \\ & R^{-1} & & \\ & & R^{-1} & \\ & & & R^{-1} \end{bmatrix}$$

$$\times \begin{bmatrix} I & & & \\ & I & & \\ & & W & \\ & & & W \end{bmatrix} \begin{bmatrix} R & & & \\ & R & & \\ & & R & \\ & & & R \end{bmatrix} \begin{bmatrix} I & & & \\ & W & & \\ & & I & \\ & & & W \end{bmatrix} \begin{bmatrix} R & & & \\ & R & & \\ & & R & \\ & & & R \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & -1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{bmatrix}$$

where $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $W = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.

This shows that any computation which can be performed in polynomial time on a classical computer, can also be performed (in principle) on a quantum computer.

However, quantum computers are also able to perform (in polynomial time) certain computations for which no polynomial-time algorithm is known. Especially:

Shor (1994) showed that an ℓ-digit number can be factored in expected time $O(\ell^2)$ on a quantum computer. (Recall: the best known algorithm on a randomized classical computer has expected execution time exponential in ℓ.)

$\mathcal{BQP}$ = the class of languages L for which there exists a polynomial-time quantum computer which accepts each word $x \in L$ with probability at least $\frac{2}{3}$, and rejects each $x \notin L$ with probability at least $\frac{2}{3}$.

It is generally believed that $\mathcal{BQP}$ contains problems in $\mathcal{NP} \setminus \mathcal{P}$ but does not contain any $\mathcal{NP}$-complete problems.