A Stroll Through the Complexity Zoo

An overview of the most important complexity classes other than P and NP

Rod Hilton

December 15, 2012

1 Introduction

Even Computer Science students very early in their careers are familiar with the "big three" complexity classes of P, NP, and NP-COMPLETE. These three classes are closely related to the biggest open questions in Computer Science, with the question of $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ being one of Clay Mathematics Institute's Millennium Prize Problems.

While these three classes are the subject of much ongoing research, the set of complexity classes is much larger than many students know. *The Complexity Zoo*[1] - an online catalog of such classes maintained by MIT faculty member Scott Joel Aaronson - currently houses 495 different complexity classes.

In this work, we will take a high-level tour of some of the most important classes which are not P, NP, or NP-COMPLETE. We will mostly focus on complexity classes that measure time complexity, and will largely ignore classes that deal with space complexity in the interest of saving space[2]. We will also focus only on decision problems rather than optimization problems or counting problems.

[1] `http://complexity-zoo.net/zoo/wiki/Complexity_Zoo`

[2] Ha!

We will see what these classes are, what kinds of languages belong to them, how they relate to other classes, what questions related to them remain unanswered, what the most recent advancements related to them are, or any other particularly interesting facts about the class. In so doing, it is the goal to provide an overview suitable for undergraduate or graduate students wishing to start a journey deeper into Complexity Theory.

2 DTIME and NTIME

A language in $\mathsf{DTIME}(f(n))$ must be recognized by some deterministic Turing machine that, given an input size n, can recognize elements in the language within $O(f(n))$ time. Many refer to this class simply as TIME.

The class DTIME, because it can be parameterized, allows us to refer to more narrowly defined time-complexity classes if they lack well-understood names. For example, we can define a complexity class for languages that can be recognized by a Turing machine in constant time, such as all even binary integers, by referring to $\mathsf{DTIME}(1)$. $\mathsf{DTIME}(logn)$ is common enough that it is often referred to as DLOGTIME.

The class P can be defined in terms of DTIME, as $\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}(n^k)$. The class EXPTIME

can be similarly defined, as $\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{n^k})$. This often expressed more simply as $\mathsf{EXPTIME} = \mathsf{DTIME}(2^{n^k})$

Closely related to this class is the Time Hierarchy Theorem, which states that $\mathsf{DTIME}(f(n))$ must be strictly contained in $\mathsf{DTIME}((f(2n+1))^3)$. This theorem is used by Hartmanis and Stearns [1965] to show that P is a proper subset of EXPTIME.

Similarly, $\mathsf{NTIME}(f(n))$ is a complexity class for languages that can be recognized by a nondeterministic Turing machine within $O(f(n))$ time. As before, $NP = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k)$ and $\mathsf{NEXPTIME} = \mathsf{NTIME}(2^{n^k})$.

NTIME has its a stronger hierarchy theorem shown in Seiferas et al. [1978], that with functions f and g and $f(n+1) = o(g)$, $\mathsf{NTIME}(f(n))$ is a strict subset of $\mathsf{NTIME}(g(n))$.

[Papadimitriou, 1994] shows P = NP, then EXPTIME = NEXPTIME, as well as if EXPTIME ≠ NEXPTIME, it would prove that P ≠ NP. It is generally accepted, however, that proving this would be even more difficult than proving P ≠ NP in the first place [Papadimitriou, 1994]. This is true as we ascend a hierarchy of exponential time classes. 2-NEXP = $\mathsf{NTIME}(2^{2^{n^k}})$, 3-NEXP = $\mathsf{NTIME}(2^{2^{2^{n^k}}})$, and so on, but if x-EXP were ever shown to not equal, x-NEXP, the inequality would apply all the way back down to EXPTIME ≠ NEXPTIME and finally P ≠ NP [Papadimitriou, 1994].

Some problems in EXPTIME include a modified version of the halting problem (does a machine halt within k steps), or evaluating a position in Chess, Checkers, or Go.

3 PP and BPP

There are a handful of classes that are defined using randomized Turing machines. A randomized Turing machine is one that has to periodically choose between two possible moves, and it flips a fair coin to decide which path to take.

A language is in PP iff there exists a randomized Turing machine that runs in polynomial time and decides membership in the language the majority of the time. Such a machine can be incorrect with a probability e so long as $e < 1/2$.

Another way of thinking about PP is to consider the set of languages that can be recognized by a nondeterministic Turing machine in polynomial time where more than half of the computation paths accept.

PP has somewhat limited usefulness because, in order to be confident in an answer from the machine, one would have to re-run the machine on the same input multiple times until the desired probability of correctness is achieved. Imagine that a nondeterministic Turing machine that recognizes a language has a million possible computation paths, and 500,001 of them accept correctly. This meets the definition required for PP, but it's extremely difficult to determine, based on the decisions of the machine alone, which half is the 'error' half. A good way to conceptualize this is to imagine being given a coin that has one slightly heavier side than the other, and your task is to determine which.

Determining which half has the slight bias requires an exponential number of experiments [Papadimitriou, 1994], so it's somewhat more useful to have a class in which the probability of error is more restrictive. BPP is just such a class.

A language is in BPP iff there exists some randomized Turing machine that runs in polynomial time and decides membership correctly with probability 3/4. The name BPP stands for Bounded-error Probabilistic Polynomial. In other words, rather than deciding by a mere majority, the Turing machine will decide by a "clear majority" [Papadimitriou, 1994].

It actually doesn't matter what probability is chosen for BPP as long as it is strictly greater than 1/2 and less than 1; any other probability within this range would still result in the same class. Imagine we were talking about any probability in this range $\frac{1}{2} + \epsilon$. The machine can be made to decide on set membership $2k + 1$ times, thus making the probability of a false answer $e^{-2\epsilon^2 k}$. The probability can be made as small as we wish by running the machine a polynomial number of times, so the probability itself isn't critical, but often 3/4 is used (we can reduce our error to 1/4 by letting $k = \lceil \frac{\ln 2}{e^2} \rceil$).

Since all of these probabilities for BPP are greater than 1/2, it's clear that BPP $\subseteq$ PP. Furthermore, since a clear majority is required in both the cases where the Turing machine correctly accepts as well as where it correctly rejects, BPP is closed under complement (BPP = CO-BPP). Since P is just BPP with an error rate of 0, P $\subseteq$ BPP.

What is not yet known is whether BPP $\subseteq$ P and thus if BPP = P. For many years, the go-to example of a language in BPP that was not known to be in P was determining whether a number was prime. One can test for primality in a probabilistic way using the Miller-Rabin primality test or the Solovay-Strassen test, both of which can be run as many times as needed to achieve a particular desired confidence in the answer, showing PRIMES to clearly be in BPP.

However, in 2002, Agrawal et al. [2004] showed that PRIMES is in P, reducing the number of problems known to be in BPP that are not also in P. It is not yet known if P = BPP, though it is strongly suspected to be the case as the number of problems known to be in BPP but not known to be in P has been decreasing.

If P = BPP, it would mean that randomized Turing machines cannot make it possible to solve NP-COMPLETE problems in polynomial time (unless P = NP). It is not currently known if BPP $\subset$ NP or NP $\subset$ BPP [Rich, 2008].

4 RP, CO-RP, and ZPP

PP and BPP both have tolerances for false negatives as well as false positives. These are referred to as having "two-sided" error. However, it is possible to build machines that have "one-sided" errors - meaning they can have errors in only one direction - which define RP and CO-RP.

A language is in RP if there is a randomized Turing machine, similar to the ones used to define PP and BPP, that runs in polynomial time and accepts inputs with probability $> 1/2/$, but rejects with a probability of 1. In other words, when the input is not in the language, the machine will always reject. When it is, the machine will be right more often than not. It can incorrectly reject, but it will never incorrectly accept.

CO-RP is the opposite, it can incorrectly accept but it will never incorrectly reject. If an input is in the language in RP, the machine will always accept, and if it is not it will reject with a probability greater than 1/2.

A language in CO-RP that has not yet been shown to be in P is that of Polynomial Identity Testing, the set of arithmetic expressions is identical to the zero-polynomial.

Finally, there is ZPP, proposed by Gill [1977]. For a language to be in ZPP, a randomized Turing machine must exist that decides "Yes", "No", or "Don't Know"; it must always return either the correct answer or "Don't Know" (it can never be wrong), and it must return "Don't Know" with a probability less than 1/2. ZPP is often defined as RP $\cap$ CO-RP.

P $\subseteq$ ZPP but it is still an open problem if ZPP = P. We know P $\subseteq$ ZPP and ZPP $\subseteq$ RP $\subseteq$ NP and ZPP $\subseteq$ CO-RP $\subseteq$ CO-NP [Rich, 2008].

An NP-COMPLETE problem being found in RP

would impliy that NP = RP which is, though theoretically possible, considered very unlikely [Moret, 1998]. In other words, RP is likely not the magic bullet for solving extremely hard problems.

Interestingly, despite the fact that the corresponding question for nondeterminism (P = NP∩CO-NP), we know that ZPP = RP∩CO-RP [Arora, 2009].

5 MA and AM

Two interesting complexity classes make use of an 'Merlin-Arthur' protocol. Imagine two players of a game, Arthur (a normal person) and Merlin (a powerful wizard). Given an instance of some problem, Merlin has unlimited computational power at his disposal, but cannot be trusted to be telling the truth. In other words, Merlin is an Oracle that can lie. Because he can lie, Arthur must verify whatever Merlin tells him, but he has the power of a BPP machine. In an Merlin-Arthur protocol, Merlin provides a polynomial proof that the answer to a problem is 'yes', and Arthur must verify it. If the answer really is 'yes', Arthur must accept with a probability of more than 2/3, and if 'no' he must reject with a probability of more than 1/3.

Another way of thinking about the Merlin-Arthur protocol is by considering it as a more random version of NP, where the oracle can lie and the machine using it must be probabilistically certain of its answer. As such, NP ⊆ MA. Arthur can ignore Merlin and solve the problem using his BPP machine, so BPP ⊆ MA. Further, Vereschchagin [1992] shows that MA ⊆ PP.

MA can be made more powerful by modifying the protocol a bit. Imagine that Arthur were to pre-flip all of the coins he would need to in order to determine his computation path through his BPP machine. This can be done without loss of generality because the sequence of coin-flips is independent of the computation itself, so it can be done ahead of time. Now, imagine that Arthur sends his sequence of coin-flips to Merlin before Merlin gives his answer. This modified protocol, the Arthur-Merlin (notice the reversed order) describes the class AM. In fact, you can have k rounds of interaction between Arthur and Merlin to describe AM(k)

Babai [1988] showed that, for any k, AM(k) = AM. Obviously, MA ⊆ AM because Arthur can get a response from Merlin, flip all of his coins, send again, and ignore Merlin's response. It remains an open question whether AM = MA.

An example of a problem in AM is graph non-isomorphism, determining if two graphs G_1 and G_2 can *not* have their nodes reordered such that they are identical. This problem is not known to be in NP, but Merlin could convince Arthur that it is so. Imagine Arthur has two graphs G_1 and G_2. He reorders one of these graphs, let's say G_1, at random, to get H, then sends H to Merlin, who must respond by declaring whether G_1 or G_2 was the graph that was reshuffled into H. If the two graphs are isomorphic, Merlin has no way of knowing which one was shuffled into H, so his guesses will be random. However, if the two graphs are actually isomorphic, Merlin will have the same answer no matter how many times the protocol is repeated. Thus, if Merlin answers consistently after a sufficient number of repetitions, Arthur can be confident that the graphs are not isomorphic.

These classes both have related classes, $\mathsf{MA}_{\mathsf{EXP}}$ and $\mathsf{AM}_{\mathsf{EXP}}$, which modify the classes so that Arthur can run in exponential time rather than polynomial, and messages between the participants can be exponentially long.

6 P/POLY

P/POLY is defined as set of languages which can be decided by Turing machines in polynomial time, when given "advice strings". An advice string is a string of input whose lengths are polynomial of the machine input, but whose contents do not depend on the input itself, and can be chosen however needed to help the algorithm decide. The reason this advice string is polynomially bounded is that, if it wasn't, any language could be decided simply by giving as an advice string a gigantic exponentially-sized lookup table of answers.

Another way P/POLY is often defined is as a class of languages that can be decided by polynomial-size circuits, which can be imagined as graphs with n inputs and one output, where each node represents a boolean operation, and whose number of nodes is a polynomial of n.

If NP $\not\subset$ P/POLY, then P $\neq$ NP [Karp and Lipton, 1980]. For many years, attempts were made to prove P $\neq$ NP by proving that NP $\not\subset$ P/POLY but so far there has been no success.

If NP $\subseteq$ P/POLY, then MA = AM. Furthermore, if EXPTIME $\subseteq$ P/POLY, then EXPTIME = AM and if NEXPTIME $\subseteq$ P/POLY, then EXPTIME = NEXPTIME = MA.

P/POLY has many cryptographic uses, with the security of an encyption scheme being analyzed by treating the adversaries as P/POLY instances. Since BPP $\subseteq$ P/POLY, doing this allows cryptologists to assume adversaries have access to the very practical BPP, as well as access to massive (but bounded) precomputed data, such as rainbow tables (a huge table for reversing 1-way hashes).

P/POLY is particularly interesting because it can decide some languages that are undecidable. To see this, we must first define a unary language, which is any language in the form f^k where f is some fixed symbol that doesn't matter. What matters is what we define k to be in the language, such as "k is prime". Every language can be mapped to a set of numbers, which means every language A has a unary version of it $(1^k|k \in A)$. There's also a complexity class for all of these kinds of languages, TALLY.

UHALT is the unary version of the halting problem, which is fair since every language has a unary version. Define UHALT = $\{1^n|n$'s binary expansion encodes a pair $\langle M, x\rangle$ such that M halts on $x\}$. P/POLY is non-uniform, which means the circuit it uses can be chosen in any way as long as it's the same for every length. So we can choose a circuit for each n by treating n as the circuit length. If $n \in HALT$, then the circuit will be true for an input w iff w is all 1's of length n. Since this can be implemented by a circuit polynomial to size n, this is fair to do, and thus P/POLY contains UHALT. This is an incredibly surprising fact, the key to understanding it is to realize that we can pull the circuit out of a magic hat that gives us whatever circuit we want as long as it varies only with the input length [Arora, 2009].

7 NC and AC

Similar to P/POLY, there are a number of other classes that deal with circuit complexity. For any d, NC^d is the set of languages that can be decided by a family of circuits C_n where C_n is a polynomial of n and is no deeper than $O(log^d n)$. The class NC^3 is the union of all such NC^d's for all integers d.

The class AC is similar, with AC^d defined the same as NC^i except that the OR and AND gates can have more than 2 bits (the fan-in) as input. AC is, of course, the union of all AC^d's.

Because fan-in on an AND or OR gate that

[3] NC stands for Nick's Class, named after Nick Pippenger by Stephen Cook

is polynomial of n can be simulated by a tree of depth $O(logn)$, the interesting property arises that, for any i, $\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NC}^{i+1}$. Furthermore, since for any i, $\mathsf{NC}^i \subseteq \mathsf{AC}^i$, it follows that $\mathsf{NC} = \mathsf{AC}$.

AC^0 is the smallest of the AC classes, it's the family of circuits that can only have a depth of O(1), but an unlimited fan-in on the gates. AC^0 is a very limited class, and it was shown by Furst et al. [1984] that the class cannot decide PARITY, the language of binary integers with an odd number of 1's. However, NC^1 does contain this language, by building circuits resembling binary trees which compute the parity of their respective halves recursively. The depth of such a tree would be $O(logn)$.

NC is generally used to characterize languages which have very efficient parallel algorithms, becuase the work of the circuits can be split and done independently, as in the PARITY example. In the same way that problems are P are considered tractable for computers, problems in NC are considered tractable for parallel computers [Homer and Selman, 2011]. It contains problems of integer addition, multiplication, division, matrix multiplication and inversing, and many more problems which are considered highly parallelizable.

It is not yet known if $\mathsf{P} = \mathsf{NC}$, in other words, if any highly parallelizable problem can be solved in polynomial time without parallelization. It is generally suspected that the answer to this question is negative [Arora, 2009].

8 BQP and QMA

The class BQP is the set of languages recognizable by a quantum Turing machine, with at most 1/3 chance of error. It is, effectively, the quantum version of BPP. A quantum Turing machine works like a regular Turing machine, except that the tape and the read-write device are not restricted to values of only 0 and 1, but can also be in a superposition of 0 and 1 and everything in between. So while a regular Turing machine can only make one calculation at a time, a quantum Turing machine can make many calculations all at once.

There is also the class QMA, which is the quantum analogue of NP or MA. It's the set of languages for which, when an element is in the language, there is a polynomial-size quantum proof that can convince a verifier (Arthur) that this is the case with a high probability (as well as when an element is not in the language).

$\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{BQP} \subseteq \mathsf{PP}$ and $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{MA} \subseteq \mathsf{QMA} \subseteq \mathsf{PP}$. BQP is generally considered the class of feasible problems on quantum computers, like P and BPP, while QMA is generally considered probably infeasible. Scott Aaronson has argued (but not proved) that BQP does not contain NP, which is to say, quantum computers cannot be used to solve NP-COMPLETE problems.

9 PH

Let's define three new classes, all equal to P. $\mathsf{P} = \Delta_0 P = \Sigma_0 P = \Pi_0 P$. Each of these classes will be used as oracles in a new set of classes, adding power to the class. So $\Delta_1 P = \mathsf{P}$ with a $\Sigma_0 P$ oracle, $\Sigma_1 P = \mathsf{NP}$ with a $\Sigma_0 P$ oracle, and $\Pi_1 P =$ CO-NP with a $\Sigma_0 P$ oracle. The first level of this hierarchy is another way to refer to the more familiar complexity classes, as $\Delta_1 P = \mathsf{P}$, $\Sigma_1 P = \mathsf{NP}$, and $\Pi_1 P =$ CO-NP More generally:

- $\Delta_i P = \mathsf{P}$ with a $\Sigma_{i-1} P$ oracle
- $\Sigma_i P = \mathsf{NP}$ with a $\Sigma_{i-1} P$ oracle
- $\Pi_i P =$ CO-NP with a $\Sigma_{i-1} P$ oracle

If we take the union of all of these classes, for all i, the resulting class is PH, the Polynomial-Time Hierarchy class.

Schaefer and Umans [2002] has a massive list, updated regularly (latest update in 2008), of different problems and which particular class inside of PH they belong. For example, MIN-MAX CLIQUE is in $\Pi_2 P$, while CLIQUE COLORING is in $\Sigma_2 P$. The problem of MINIMUM CIRCUIT, which determines if a boolean circuit can be represented by an equivalent circuit with fewer gates, is $\Pi_2 P$.

Toda [1989] shows that PH $\subseteq$ P with a PP oracle . If P = NP then the entire PH hierarchy collapses to P. $\Sigma_2 P$ contains AM.

Without proof that P $\neq$ NP, Papadimitriou [1994] argues that it is impossible to show that each level of the hierarchy properly contains the next. We know that PH $\subseteq$ PSPACE, but it is an open question if PH = PSPACE. If it were the case that PH = PSPACE, then once again the entire PH hierarchy would collapse [Hemaspaandra and Ogihara, 2001].

10 Remarks

We've looked at many complexity classes here, including AC^0, AC, AM, BPP, BQP, DLOGTIME, DTIME, EXPTIME, MA, NC, NEXPTIME, NTIME, P/POLY, PH, PP, PSPACE, RP, QMA, TALLY, ZPP, CO-BPP, CO-NP, CO-RP, and of course P, NP, and NP-COMPLETE. Still, that's only 26 classes, leaving over 450 more classes in the Complexity Zoo that haven't even been mentioned. Many of these classes are small tweaks and variations on those covered, but Complexity Theory is still an immense field.

These many classes often share tight relationships with other classes, with a single equivalence or containment proof kicking off a series of collapsing hierarchies or equalities. And of course, many of these classes have open problems whose solutions would lead directly to solving the white whale of Complexity Theory, P $\stackrel{?}{=}$ NP.

Scott Aaronson, keeper of the Complexity Zoo, published a post on his blog a few years ago entitled "Eight Signs A Claimed P $\neq$ NP Proof Is Wrong"[4]. In the post, Dr. Aaronson implores readers not to dismiss a proof based on the author's credentials or background, but to challenge the proofs when they fail to address eight specific points. Nearly all eight of these points involve an author not explaining how his or her proof interplays with various complexity classes and languages. The third, for example, states:

> The paper doesn't prove any weaker results along the way: for example, P $\neq$ PSPACE, NEXP $\not\subset$ P/POLY ... P vs. NP is a staggeringly hard problem, which one should think of as being dozens of steps beyond anything that we know how to prove today.

Given how important these varied complexity classes may prove to be in the process of chipping away at the P $\stackrel{?}{=}$ NP question, it's certainly beneficial to study and understand them, and The Complexity Zoo is an indispensable resource.

References

M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004.

S. Arora. *Computational complexity : a modern approach*. Cambridge University Press, Cambridge New York, 2009. ISBN 9780521424264.

L. Babai. Arthur-merlin games: A randomized proof system and a hierarchy of complexity

[4] http://www.scottaaronson.com/blog/?p=458

classes. *J. Comput. Syst. Sci.*, 36:254–276, 1988.

M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Theory of Computing Systems*, 17(1):13–27, 1984.

J. Gill. Computational complexity of probabilistic turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, pages 285–306, 1965.

L. Hemaspaandra and M. Ogihara. *The complexity theory companion*. Springer, 2001.

S. Homer and A. Selman. *Computability and complexity theory*. Springer, New York, 2011. ISBN 9781461406815.

R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 302–309. ACM, 1980.

B. Moret. *The theory of computation*. Addison-Wesley, Reading, Mass, 1998. ISBN 9780201258288.

C. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Mass, 1994. ISBN 9780201530827.

E. Rich. *Automata, computability and complexity : theory and applications*. Pearson Prentice Hall, Upper Saddle River, N.J, 2008. ISBN 9780132288064.

M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT news*, 33(3):32–49, 2002.

J. Seiferas, M. Fischer, and A. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM (JACM)*, 25(1):146–167, 1978.

S. Toda. On the computational power of pp and p. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 514–519. IEEE, 1989.

N. Vereschchagin. On the power of pp. In *Structure in Complexity Theory Conference, 1992., Proceedings of the Seventh Annual*, pages 138–143. IEEE, 1992.