Complexity Theory
Complexity

Charles E. Hughes

COT6410 – Spring 2022 Notes

Graph Coloring

- Instance: A graph **G = (V, E)** and an integer **k**.
- Question: Can **G** be "properly colored" with at most **k** colors?

- Proper Coloring: one of the **k** colors is assigned to each vertex so that adjacent vertices have different colors.

- Suppose we have two instances of this problem (1) is True (Yes) and the other (2) is False (No).

- AND, you know (1) is Yes and (2) is No. (Maybe you have a secret program that has analyzed the two instance.)

Checking a “Yes” Answer

- Without showing how your program works (you may not even know), how can you convince someone else that instance (1) is, in fact, a Yes instance?
- We can assume the output of the program was an actual coloring of **G**. Just give that to a doubter who can easily check that no adjacent vertices are colored the same, and that no more than **k** colors were used.
- How about the No instance?
- What could the program have given that allows us to quickly "verify" (2) is a No instance?
    - No One Knows!!
- For all problems seem to be harder than there exists ones in many contexts

Checking a "No" Answer

- The only thing anyone has thought of is to have it test all possible ways to **k**-color the graph – all of which fail, of course, if "No" is the correct answer.
- There are an exponential number of things (colorings) to check.
- For some problems, there seems to be a big difference between verifying Yes and No instances.
- To solve a problem efficiently, we must be able to solve both Yes and No instances efficiently and so it would seem we should be able to verify both quickly.

Hard and Easy

- True Conjecture: If a problem is easy to solve, then it is easy to verify (just solve it and compare).
- Contrapositive: If a problem is hard to verify, then it is (probably) hard to solve.
- There is nothing magical about Yes and No instances – sometimes the Yes instances are hard to verify and No instances are easy to verify.
- And, of course, sometimes both are hard to verify.

Easy Verification

- Are there problems in which both Yes and No instances are easy to verify?

- Yes. For example: Search a list **L** of **n** values for a key **x**.
- Question: Is **x** in the list **L**?

- Yes and No instances are both easy to verify.

- In fact, the entire problem is easy to solve!!

Verify vs Solve

- Conjecture: If both Yes and No instances are easy to verify, then the problem is easy to solve.
- No one has yet proven this claim, but most researchers believe it to be true.
- Note: It is usually relatively easy to prove something is easy – just write an algorithm for it and prove it is correct and that it is fast (usually, we mean polynomial).
- But, it is usually very difficult to prove something is hard – we may not be clever enough yet. So, you will often see "appears to be hard."

A CS Grand Challenge

Does *P=NP*?

There are many equivalent ways to describe *P* and *NP*. For now, we will use the following.

P is the set of decision problems (those whose instances have "Yes"/ "No" answers) that can be solved in polynomial time on a deterministic computer (no concurrency or guesses allowed).

NP is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn an unbounded number of parallel threads; equivalently one that can be verified in polynomial time on a deterministic computer).

Again, as "Does ***P=NP*?**" has just one question, it is solvable, we just don't yet know which solution, "Yes" or "No", is the correct one.

ORDER ANALYSIS

Notion of “Order”

Throughout the complexity portion of this course, we will be interested in how long an algorithm takes on the instances of some arbitrary "size" **n**. Recognizing that different times can be recorded for two instance of size **n**, we only ask about the worst case.

We also understand that different languages, computers, and even skill of the implementer can alter the "running time."

Notion of “Order”

As a result, we really can never know "exactly" how long anything takes.

So, we usually settle for a substitute function, and say the function we are trying to measure is "of the order of" this new substitute function.

Notion of “Order”

"Order" is something we use to describe an upper bound upon something else (in our case, time, but it can apply to almost anything).

For example, let **f(n)** and **g(n)** be two functions. We say "**f(n)** is order **g(n)**" when there exists constants **c** and **N** such that **f(n) ≤ cg(n)** for all **n ≥ N**.

What this is saying is that when **n** is 'large enough,' **f(n)** is bounded above by a constant multiple of **g(n)**.

Notion of “Order”

This is particularly useful when **f(n)** is not known precisely, is complicated to compute, and/or difficult to use. We can, by this, replace **f(n)** by **g(n)** and know we aren't "off too far."

We say **f(n)** is "in the order of **g(n)**" or, simply, **f(n)** $\in$ **O(g(n))**.

Usually, **g(n)** is a simple function, like **nlog(n)**, $\mathbf{n^3}$, $\mathbf{2^n}$, etc., that's easy to understand and use.

Notion of "Order"

Order of an Algorithm: The maximum number of steps required to find the answer to any instance of size **n**, for any arbitrary value of **n**.

For example, if an algorithm requires at most $\mathbf{6n^2+3n-6}$ steps on any instance of size n, we say it is "order $\mathbf{n^2}$" or, simply, $\mathbf{O(n^2)}$.

Order

Let the order of algorithm **X** be in $\mathbf{O(f_x(n))}$.

Then, for algorithms **A** and **B** and their respective order functions, $\mathbf{f_A(n)}$ and $\mathbf{f_B(n)}$, consider the limit of $\mathbf{f_A(n)/f_B(n)}$ as **n** goes to infinity.

If this value is

0	**A** is faster than **B**
constant	**A** and **B** are "equally slow/fast"
infinity	**A** is slower than **B**.

Order of a Problem

Order of a Problem

The order of the fastest algorithm that can ever solve this problem. (Also known as the "Complexity" of the problem.)

Often difficult to determine, since this allows for algorithms not yet discovered.

Decision vs Optimization

Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)

Vertex Cover (VC)

- Suppose we are in charge of a large network (a graph where edges are links between pairs of cities (vertices). Periodically, a line fails. To mend the line, we must call in a repair crew that goes over the line to fix it. To minimize down time, we station a repair crew at one end of every line. How many crews must you have and where should they be stationed?

- This is called the Vertex Cover Problem. (Yes, it sounds like it should be called the Edge Cover problem – something else already had that name.)

- An interesting problem – it is among the hardest problems, yet is one of the easiest of the hard problems.

VC Decision vs Optimization

- As a Decision Problem:

- Instances: A graph **G** and an integer **k**.
- Question: Does **G** possess a vertex Cover with at most **k** vertices?

- As an Optimization Problem:

- Instances: A graph **G**.
- Question: What is the smallest **k** for which **G** possesses a vertex cover?

Relation of VC Problems

- If we can (easily) solve either one of these problems, we can (easily) solve the other. (To solve the optimization version, just solve the decision version with several different values of **k**. Use a binary search on **k** between **1** and **n**. That is **log(n)** solutions of the decision problem solves the optimization problem. It's simple to solve the decision version if we can solve the optimization version.

- We say their time complexity differs by no more than a multiple of **log(n)**.
- If one is polynomial then so is the other.
- If one is exponential, then so is the other.

- We say they are equally difficult (both poly. or both exponential).

Smallest VC

- A "stranger version"

- Instances: A graph **G** and an integer **k**.
- Question: Does the smallest vertex cover of **G** have exactly **k** vertices?
- This is a decision problem. But, notice that it does not seem to be easy to verify either Yes or No instances!! (We can easily verify No instances for which the VC number is less than **k**, but not when it is actually greater than **k**.)

- So, it would seem to be in a different category than either of the other two. Yet, it also has the property that if we can easily solve either of the first two versions, we can easily solve this one.

Natural Pairs of Problems

Interestingly, these usually come in pairs

a *decision* problem, and

an *optimization* problem.

Equally easy, or equally difficult, to solve.

Both can be solved in polynomial time, or both require exponential time.

A Word about Time

An algorithm for a problem is said to be polynomial if there exists integers **k** and **N** such that **t(n)**, the maximum number of steps required on any instance of size **n**, is at most $\mathbf{n^k}$, for all $\mathbf{n \geq N}$.

Otherwise, we say the algorithm is exponential. Usually, this is interpreted to mean $\mathbf{t(n) \geq c^n}$ for an infinite set of size **n** instances, and some constant $\mathbf{c > 1}$ (often, we simply use $\mathbf{c = 2}$).

A Word about “Words”

Normally, when we say a problem is "easy" we mean that it has a polynomial algorithm.

But, when we say a problem is "hard" or “apparently hard" we usually mean no polynomial algorithm is known, and none seems likely.

It is possible a polynomial algorithm exists for "hard" problems, but the evidence seems to indicate otherwise.

A Word about Abstractions

Problems we will discuss are usually "abstractions" of real problems. That is, to the extent possible, non-essential features have been removed, others have been simplified and given variable names, relationships have been replaced with mathematical equations and/or inequalities, etc.

If an abstraction is hard, then the real problem is probably even harder!!

A Word about Toy Problems

This process, Mathematical Modeling, is a field of study in itself, and not our interest here.

On the other hand, we sometimes conjure up artificial problems to put a little "reality" into our work. This results in what some call "toy problems."

Again, if a toy problem is hard, then the real problem is probably harder.

Very Hard Problems

Some problems have no algorithm (e. g., Halting Problem.)

No mechanical/logical procedure will ever solve all instances of any such problem!!

Some problems have only exponential algorithms (provably so – they must take at least order $\mathbf{2^n}$ steps) So far, only a few have been proven, but there may be many. We suspect so.

Easy Problems

Many problems have polynomial algorithms (Fortunately).

Why fortunately? Because, most exponential algorithms are essentially useless for problem instances with **n** much larger than 50 or 60. We have algorithms for them, but the best of these will take 100's of years to run, even on much faster computers than we now envision.

Three Classes of Problems

Problems proven to be in these three groups (classes) are, respectively,

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes.

Unknown Complexity

Practically, there are a lot of problems (maybe, most) that have not been proven to be in any of the classes (Yet, maybe never will be).

Most currently "lie between" polynomial and exponential – we know of exponential algorithms, but have been unable to prove that exponential algorithms are necessary.

Some may have polynomial algorithms, but we have not yet been clever enough to discover them.

Why do we Care?

If an algorithm is $\mathbf{O(n^k)}$, increasing the size of an instance by one gives a running time that is $\mathbf{O((n+1)^k)}$

That's really not much more.

With an increase of one in an exponential algorithm, $\mathbf{O(2^n)}$ changes to $\mathbf{O(2^{n+1}) = O(2*2^n) = 2*O(2^n)}$ – that is, it takes about twice as long.

A Word about “Size”

Technically, the size of an instance is the minimum number of bits (information) needed to represent the instance – its "length."

This comes from early Formal Language researchers who were analyzing the time needed to 'recognize' a string of characters as a function of its length (number of characters).

When dealing with more general problems there is usually a parameter (number of vertices, processors, variables, etc.) that is polynomially related to the length of the instance. Then, we are justified in using the parameter as a measure of the length (size), since anything polynomially related to one will be polynomially related to the other.

The Subtlety of “Size”

But, be careful.

For instance, if the "value" (magnitude) of n is both the input and the parameter, the 'length' of the input (number of bits) is $\log_2(n)$. So, an algorithm that takes n time is running in $n = 2^{\log2(n)}$ time, which is exponential in terms of the length, $\log_2(n)$, but linear (hence, polynomial) in terms of the "value," or magnitude, of n.

It's a subtle, and usually unimportant difference, but it can bite you.

Subset Sum

- Problem – Subset Sum

- Instances: A list **L** of **n** integer values and an integer **B**.
- Question: Does **L** have a subset which sums exactly to **B**?

- No one knows of a polynomial (deterministic) solution to this problem.

- On the other hand, there is a very simple (dynamic programming) algorithm that runs in **O(nB)** time.

- Why isn't this "polynomial"?
- Because, the "length" of an instance is **nlog(B)** and
- **nB > (nlog(B))^k** for any fixed **k**.

Why do we Care?

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution.

You should know something about how hard a problem is before you try to solve it.

Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

Algorithms for any of these
(area of Algorithm Design/Analysis)

Complexity Theory

Second Part of Course

Models of Computation

NonDeterminism

Since we can't seem to find a model of computation that is more powerful than a TM, can we find one that is 'faster'?

In particular, we want one that takes us from exponential time to polynomial time.

Our candidate will be the NonDeterministic Turing Machine (NDTM).

NDTM's

In the basic Deterministic Turing Machine (DTM) we make one major alteration (and take care of a few repercussions):

- **The 'transition functon' in DTM's is allowed to become a 'transition mapping' in NDTM's.**

This means that rather than the next action being totally specified (deterministic) by the current state and input character, we now can have many next actions - simultaneously. That is, a NDTM can be in many states at once. (That raises some interesting problems with writing on the tape, just where the tape head is, etc., but those little things can be explained away).

NDTM's

We also require that there be only one halt state - the 'accept' state. That also raises an interesting question - what if we give it an instance that is not 'acceptable'? The answer - it blows up (or goes into an infinite loop).

The solution is that we are only allowed to give it 'acceptable' input. That means

NDTM's are only defined for decision problems and, in particular, only for Yes instances.

NDTM's

We want to determine how long it takes to get to the accept state - that's our only motive!!

So, what is a NDTM doing?

In a normal (deterministic) algorithm, we often have a loop where each time through the loop we are testing a different option to see if that "choice" leads to a correct solution. If one does, fine, we go on to another part of the problem. If one doesn't, we return to the same place and make a different choice, and test it, etc.

NDTM's

If this is a Yes instance, we are guaranteed that an acceptable choice will eventually be found and we go on.

In a NDTM, what we are doing is making, and testing, all of those choices at once by 'spawning' a different NDTM for each of them. Those that don't work out, simply die (or something).

This is kind of like the ultimate in parallel programming.

NDTM's

To allay concerns about not being able to write on the tape, we can allow each spawned NDTM to have its own copy of the tape with a read/write head.

The restriction is that nothing can be reported back except that the accept state was reached.

NDTM's

Another interpretation of nondeterminism:

From the basic definition, we notice that out of every state having a nondeterministic choice, at least one choice is valid and all the rest sort of die off. That is they really have no reason for being spawned (for this instance - maybe for another). So, we station at each such state, an 'oracle' (an all knowing being) who only allows the correct NDTM to be spawned.

An 'Oracle Machine.'

NDTM's

This is not totally unreasonable. We can look at a non deterministic decision as a deterministic algorithm in which, when an "option" is to be tested, it is very lucky, or clever, to make the correct choice the first time.

In this sense, the two machines would work identically, and we are just asking "How long does a DTM take if it always makes the correct decisions?"

NDTM's

As long as we are talking magic, we might as well talk about a 'super' oracle stationed at the start state (and get rid of the rest of the oracles) whose task is to examine the given instance and simply tell you what sequence of transitions needs to be executed to reach the accept state.

He/she will write them to the left of cell 0 (the instance is to the right).

NDTM's

Now, you simply write a DTM to run back and forth between the left of the tape to get the 'next action' and then go back to the right half to examine the NDTM and instance to verify that the provided transition is a valid next action. As predicted by the oracle, the DTM will see that the NDTM would reach the accept state and can report the number of steps required.

NDTM's

All of this was originally designed with Language Recognition problems in mind. It is not a far stretch to realize the Yes instances of any of our more real word-like decision problems defines a language, and that the same approach can be used to "solve" them.

Rather than the oracle placing the sequence of transitions on the tape, we ask him/her to provide a 'witness' to (a 'proof' of) the correctness of the instance.

NDTM's

For example, in the SubsetSum problem, we ask the oracle to write down the subset of objects whose sum is B (the desired sum). Then we ask "Can we write a deterministic polynomial algorithm to test the given witness."

The answer for SubsetSum is Yes, we can, i.e., the witness is verifiable in deterministic polynomial time.

NDTM's - Witnesses

Just what can we ask and expect of a "witness"?

The witness must be something that

(1) we can verify to be accurate (for the given problem and instance) and

(2) we must be able to "finish off" the solution.

All in polynomial time.

NDTM's - Witnesses

The witness can be nothing!

Then, we are on our own. We have to "solve the instance in polynomial time."

The witness can be "Yes."

Duh. We already knew that. We have to now verify the yes instance is a yes instance (same as above).

The witness has to be something other than nothing and Yes.

NDTM's - Witnesses

The information provided must be something we could have come up with ourselves, but probably at an exponential cost. And, it has to be enough so that we can conclude the final answer Yes from it.

Consider a witness for the graph coloring problem:

Given: A graph G = (V, E) and an integer k.

Question: Can the vertices of G be assigned colors so that adjacent vertices have different colors and use at most k colors?

NDTM's - Witnesses

The witness could be nothing, or Yes.

But that's not good enough - we don't know of a polynomial algorithm for graph coloring.

It could be "vertex 10 is colored Red."

That's not good enough either. Any single vertex can be colored any color we want.

It could be a color assigned to each vertex.

That would work, because we can verify its validity in polynomial time, and we can conclude the correct answer of Yes.

NDTM's - Witnesses

What if it was a color for all vertices but one?

That also is enough. We can verify the correctness of the n-1 given to us, then we can verify that the one uncolored vertex can be colored with a color not on any neighbor, and that the total is not more than k.

What if all but 2, 3, or 20 vertices are colored

All are valid witnesses.

What if half the vertices are colored?

Usually, No. There's not enough information. Sure, we can check that what is given to us is properly colored, but we don't know how to "finish it off."

NDTM's - Witnesses

An interesting question: For a given problem, what are the limits to what can be provided that still allows a polynomial verification?

NDTM's

A major question remains: Do we have, in NDTMs, a model of computation that solves all deterministic exponential (DE) problems in polynomial time (nondeterministic polynomial time)??

It definitely solves some problems we *think* are DE in nondeterministic polynomial time.

NDTM's

But, so far, all problems that have been proven to require deterministic exponential time also require nondeterministic exponential time.

So, the jury is still out. In the meantime, NDTMs are still valuable, because they might identify a larger class of problems than does a deterministic TM - the set of decision problems for which Yes instances can be verified in polynomial time.

Problem Classes

We now begin to discuss several different classes of problems. The first two will be:

NP	**'Nondeterministic' Polynomial**
P	**'Deterministic' Polynomial,**
	The 'easiest' problems in NP

Their definitions are rooted in the depths of Computability Theory as just described, but it is worth repeating some of it in the next few slides.

Problem Classes

We assume knowledge of Deterministic and Nondeterministic Turing Machines. (DTM's and NDTM's)

The only use in life of a NDTM is to scan a string of characters X and proceed by state transitions until an 'accept' state is entered.

X must be in the language the NDTM is designed to recognize. Otherwise, it blows up!!

Problem Classes

So, what good is it?

We can count the number of transitions on the shortest path (elapsed time) to the accept state!!!

If there is a constant k for which the number of transitions is at most $|X|^k$, then the language is said to be 'nondeterministic polynomial.'

Problem Classes

The subset of YES instances of the set of instances of a decision problem, as we have described them above, is a language.

When given an instance, we want to know that it is in the subset of Yes instances. (All answers to Yes instances look alike - we don't care which one we get or how it was obtained).

This begs the question "What about the No instances?"

The answer is that we will get to them later. (They will actually form another class of problems.)

Problem Classes

This actually defines our first Class, NP, the set of decision problems whose Yes instances can be solved by a Nondeterministic Turing Machine in polynomial time.

That knowledge is not of much use!! We still don't know how to tell (easily) if a problem is in NP. And, that's our goal.

Fortunately, all we are doing with a NDTM is tracing the correct path to the accept state. Since all we are interested in doing is counting its length, if someone just gave us the correct path and we followed it, we could learn the same thing - how long it is.

Problem Classes

It is even simpler than that (all this has been proven mathematically). Consider the following problem:

> **You have a big van that can carry 10,000 lbs. You also have a batch of objects with weights w_1, w_2, …, w_n lbs. Their total sum is more than 10,000 lbs, so you can't haul all of them.**
>
> **Can you load the van with exactly 10,000 lbs?**
>
> **(WOW. That's the SubsetSum problem.)**

Problem Classes

Now, suppose it is possible (i.e., a Yes instance) and someone tells you exactly what objects to select.

We can add the weights of those selected objects and verify the correctness of the selection.

This is the same as following the correct path in a NDTM. (Well, not just the same, but it can be proven to be equivalent.)

Therefore, all we have to do is count how long it takes to verify that a "correct" answer" is in fact correct.