Great Ideas in Computing

University of Toronto CSC196
Fall 2021

Week 9: November 14-18 (2022)

Announcements

Announcements

- I have posted all of Assignment 3 (A3) on the web page. A3 is due Friday, Oct 18 at 8AM.
- Any questions on assignment? Would you prefer a class on Wed (and still I am able to answer questions on assignment) or tutorial on Wed?
- The next and final quiz is scheduled for November 25.

This weeks agenda

Agenda

- We ended the Wednesday, November 2 class on slide 34 with a discussion of the extended Church-Turing thesis. I will make one final comment about Quantum computing and the impact on the extended Church Turing thesis.
- We will define the classes P and NP, NP-completeness and the $P \neq NP$ conjecture and its importance.
- Then (probably next week) we begin complexity based cryptography.

But what if quantum computers become practical?

Lets assume the quantum computers or other non-classical computers become practical. We are about to discuss the P vs NP issue and the $P \neq NP$ conjecture, the central question in complexity theory.

This conjecture is formulated with respect to the extended Turing thesis. That is, we are accepting the definition that "efficiently computable" means polynomial time computable by a Turing machine. Will everything about this question and conjecture become useless if we someday have available more powerful non-classical (e.g., quantum) computers?

But what if quantum computers become practical?

Lets assume the quantum computers or other non-classical computers become practical. We are about to discuss the P vs NP issue and the $P \neq NP$ conjecture, the central question in complexity theory.

This conjecture is formulated with respect to the extended Turing thesis. That is, we are accepting the definition that "efficiently computable" means polynomial time computable by a Turing machine. Will everything about this question and conjecture become useless if we someday have available more powerful non-classical (e.g., quantum) computers?

No, the theory we will be developing can be reformulated in terms of a new computational model. We will have new functions (like factoring integers) which will now become efficiently computable (assuming they were not efficiently computable classically). But still there will be an analogous complexity theory based on the (for now hypothetical) new computational model. Moroever, our current belief is that there are problems in the class NP that are not computable in polynomial time on a quantum computer.

Polynomial time computable decision problems

We will now restrict attention to decision problems; that is $f : \Sigma^* \rightarrow \{YES, NO\}$. Σ is a finite alphabet and Σ^* is the set of all strings over Σ. We can also identify $\{YES, NO\}$ with say $\{1, 0\}$.

Equivalently, we are considering languages $L \subseteq \Sigma^*$.

The class of languages (decision problems) P is defined as the set of languages L that are decideable in polynomial time on a Turing machine; that is the languages that are "efficiently decideable".

In what follows, I will assume we have some agreed upon way that we represent graphs $G = (V, E)$ as strings over some finite alphabet Σ. Without refering to the representation, let $L_{connected} = \{G = (V, E) | G$ is a connected graph$\}$.

It is not difficult to show that $L_{connected}$ is in the class P. (For example, we can use breadth first search.)

A language “probably not” in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in V $\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The “well-known” *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in V $\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

A language "probably not" in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in V $\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The "well-known" *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

A language “probably not” in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in V $\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The “well-known” *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

But can I efficiently prove to you that G does *not*‘ have a HC?

A language “probably not” in the class P

Consider the following language: $L_{HC} = \{G = (V, E) | G$ has a simple cycle including all nodes in V $\}$. It is strongly believed (but not proven) that L_{HC} is not polynomial time computable.

A simple cycle containing all the nodes in the graph is called a Hamiltonian cycle (HC). (The “well-known” *traveling salesman problem* (TSP) is to find an HC of least cost in an edge weighted graph. Have you heard of this problem?.)

But suppose that a given graph G has Hamiltonian cycle. How can I convince you that G has such a cycle

I can simply show you a Hamiltonian cycle C (assuming I know C) and you can easily and efficiently verify that C is indeed a HC. That is, I can prove to you that G has a HC.

But can I efficiently prove to you that G does *not*‘ have a HC?

““Probably not”

NP: the class of languages which are "efficiently verifiable"

Using the HC problem as an example, lets define what it means to be efficiently verifiable.

Let L be a language (like L_{HC}) that satisfies the following conditions: There is a polynomial time decdeable relation $R(x, y)$ and a polynomial p such that for every x, $x \in L$ if and only if there exists a y with $|y| \leq p(|x|)$ and $R(x, y) = TRUE$.

$R(x, y)$ is a verification relation (or predicate) and y is called a certificate that verifies x being in L.

The class NP is the class of languages (decision problems) that have such a verification relation and certificate.

For example HC is in NP. Namely, given a representation x of a graph $G = (V, E)$, a certificate y is an encoding of a sequence of vertices specifying a Hamiltonian cycle C . $R(x, y)$ checks the conditions for $y = C$ being a simple cycle containing all the nodes in V .

The million \$ question: Is $P \neq NP$

This is literally (and not just figuratively) a million \$ question for someone who solves the question. In fact, it is worth much more that just one million \$ for a proof that either $P = NP$ or a proof that $P \neq NP$.

Cook defined the concept of NP-completeness and gave a couple of examples of such problems, namely SAT and $CLIQUE$, problems in NP that are believed to not be in P.

We wil define NP-completeness and the evidence for the conjecture that $P \neq NP$.

The important consequence of NP completeness is that if *any* NP decision problem turns out to be in P, then $P = NP$. Since we strongly believe $P \neq NP$, this means that we strongly believe that no NP complete problem can be in P.

NP: the class of languages which are “efficiently verifiable”

Using the HC problem as an example, lets define what it means to be efficiently verifiable.

Let L be a language (like L_{HC}) that saitisfies the following conditions: There is a polynomial time decidable relation $R(x, y)$ and a polynomial p such that for every x, $x \in L$ if and only if there exists a y with $|y| \leq p(|x|)$ and $R(x, y) = TRUE$.

$R(x, y)$ is a *verification* relation (or predicate) and y is called a *certificate* with respect to R that verifies x being in L.

Definition: The class NP is the class of languages (decision problems) that have such a verification relation and certificate.

For example L_{HC} is in NP. Namely, given a representation x of a graph $G = (V, E)$, a certificate y is an encoding of a sequence of vertices specifying a Hamiltonian cycle C. $R(x, y)$ checks the conditions for C being a simple cycle containing all the nodes in V.

Many many decision problems are in the class *NP*

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Many many decision problems are in the class NP

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Consider a language L (like $L_{connected}$) that is decideable in polynomial time. Then in the definition of NP, we can let let $R(x, y)$ be the relation that is $TRUE$ iff $x \in L$ ignoring y and $R(x, y)$ is polynomial time since we can decide if $x \in L$ in polynomial time by the assumption that $L \in P$.

Many many decision problems are in the class NP

First we will note that the class P (decision problems decideable in polynomial time) is a subset of NP; that is, $P \subseteq NP$. Is this obvious?

Consider a language L (like $L_{connected}$) that is decideable in polynomial time. Then in the definition of NP, we can let let $R(x, y)$ be the relation that is $TRUE$ iff $x \in L$ ignoring y and $R(x, y)$ is polynomial time since we can decide if $x \in L$ in polynomial time by the assumption that $L \in P$.

In saying $P \subseteq NP$, we have left open the possibility that $P = NP$. However, the widely believed assumption (conjecture) is that $P \neq NP$. This question (conjecture) was implicitly asked by (for example) Gauss (early 1800's), von Neumann, Gödel (1950's) , Cobham, and Edmonds (1960s). The conjecture was formalized by Cook in 1971 (indpendently by Levin in the FSU but his work was not known until about 1973).

More specifically Cook defined the concept of NP-completeness and gave a couple of examples of such problems, namely SAT and $CLIQUE$, problems in NP that are believed to *not* be in P. We wil define NP-completeness and the evidence for the conjecture that $P \neq NP$.

Some examples of *NP* complete decision problems

In our examples we always assume some natural way to represent the inputs as strings over some finite alphabet. In particular, integers are represented in say binary or decimal. Polynomial time means time bounded by a polynomial $p(n)$ where n is the length of the input string.

I will explain each of the following decision problems as we introduce them. Some problems are naturally decision problems. Others are decision variants of optimization problems and other relations or functions. Each of these decision problems are easily seen to be in *NP* (i.e. it is easy to provide a verification proedicate and succinct certificate). We will soon define completeness and indicate why each of these problems is *NP* complete.

- L_{HC} as defined previously; i.e., the set of graphs that have a Hamiltonian cycle.
- $SAT = \{F|F$ is a propositional formula that is *satisfiable*$\}$
- $PARTITION = \{(a_1, a_2, \ldots, a_n)|\exists S : \sum_{a_i \in S} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i\}$
- $VERTEX\text{-}COLOUR = \{(G, k)|G$ can be vertex coloured with k colours$\}$

A example of a language in NP language that is believed to not be NP complete and believed to not be in P

$FACTOR = \{(N,k) | N$ is an integer that has a proper factor $m \leq k\}$

It is easy to see that $FACTOR$ is in NP.

Suppose $FACTOR \in P$. Can you see how to factor a number N (i.e. provide the prime factorization) in polynomial time?

We have mentioned before that it is widely believed that we cannot factor integers in polynomial time and we use that assumption for some cryptographic applications.

The problem of efficiently factoring goes back at least two centuries to Gauss.

We will soon see some evidence that $FACTOR$ is not complete.

NP completeness

A decison problem (or any problem) L is NP-hard if *every* problem $L' \in NP$ can be "efficicently reduced" to L. There are different notions of how to formalize "efficiently reduced" and we will discuss this shortly.

A decision problem L is NP-complete if it is both in NP and NP-hard. **Here are the immediate consequences of a problem being NP-complete.**

- If L is NP complete, and $L \in P$, then every $L' \in NP$ is in P
- Equivalently, if any $L' \in NP$ is not in P, then every NP-complete problem is not in P.
- There are hundreds (and really thousands) of problems that are NP-complete and since we *"religously"* believe $P \neq NP$, we believe that none of these complete problems can be decided in polynomial time. (I emphasize that this is in terms of *worst case* complexity.)

End of Monday, November 14 class

We ended to class having just defined the notion of a language (decision problem) being *NP*-hard and *NP*-complete.

Note: There are *NP* hard languages that are not in *NP*. In particular, there are languages that *provably* require exponential time.

We will have to discuss polynomial time reduction and polynomial time transformation, the polynomial time analogues of a Turing computable reduction and a computable transformation.

We will then be able to discuss the immediate consequences and *NP*-completeness and show how one can exhibit examples of *NP* complete sets.

Why the religious belief

Why do we believe so strongly that $P \neq NP$. It is simply that many very talented people over literally centuries have tried to efficiently solve problems that are in NP (and believed to not be in P and especially those that are NP-complete) and failed to do so.

Even so, there have been surprises in complexity theory and one still has to keep in mind that $P \neq NP$ is still a conjecture and not a proven result.

Our confidence in this conjecture is strong enough that modern day cryptography makes this assumption and indeed makes even stronger assumptions. For example, cryptographic protocols usually assume that there exist *one-way functions* f for which it is easy (i.e. poly time) to compute $f(x)$ for any x but given y, it is difficult to find an x such that $f(x) = y$.

For cryptography we also need assurance that a problem is not only hard in a worst case sense but also hard in some "average case " sense.

What would happen if someone solves the P vs NP question?

A frequent question that is asked is the following: What would be the consequences if someone resolves the P vs NP question

What would happen if someone solves the P vs NP question?

A frequent question that is asked is the following: What would be the consequences if someone resolves the P vs NP question

While the mathematical and scientfiic impact will be enormous, mathematics and science will not end.

If someoone proves that (as we do not believe) $P = NP$, then the "practical impact" will depend on how efficiently we can solve NP complete problems; that is, what are the polynomial time bounds.

If someone prove $P \neq NP$, then the "practical impact" will depend on whether or not a given problem can be solved efficiently "in practice" (i.e. for most inputs or for "the inputs we care about"). More about worst case vs "practical application" later.

Returning to the concept of reduction

At the heart of NP completeness and more generally algorithm analysis is the concept of (efficient) reduction of problems. When we say that problem A "efficiently" reduces to problem B, we can conclude that an efficient algorithm for B will result in an efficient algorithm for A (and equivalently, the contrapositive states that A not efficiently computable implies that B is not efficiently computable).

There are different definitions for what we mean by an efficient reduction and the precise definition matters in terms of what we want to conclude from the reduction.

One major distinction is between a very general type of reduction (which we will just call *poly time reduction* (i.e., the poly time version of Turing reduction) and the more restricted reduction which we will call *poly time transformation* (i.e., the polynomial time version of a Turing computable transformation).

Two types of reductions continued

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?

Two types of reductions continued

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?
It is easy to see that $A \leq_{trans}^{poly} B$ and $B \in P$ implies $A \in P$.

Following Cook's paper, Karp provided a list of 21 combinatorial and graph theoretical problems that are NP complete. Karp used the more restrictive $\leq_{trans}^{poly}$. If you like names associated with these reductions then we can denote $\leq_T^{poly}$ as $\leq_{Cook}$ and $\leq_{trans}^{poly}$ as $\leq_{Karp}$.

Comparing the reductions $\leq_T^{poly}$ and $\leq_{trans}^{poly}$

Let first explicitly give the definition NP-complete.

Definition: A language (or decision problem) L is NP complete if

1. $L \in NP$.
2. L is NP-hard *with respect to some polynomial time reduction*, for example with respect to either $\leq_T^{poly}$, or $\leq_{trans}$ $poly$. That is, if we are using $\leq_{trans}^{poly}$, then L is is NP-hard if for every $A \in NP$, there there is a polynomial time computable function h such that $w \in A$ if and only if $h(w) \in L$.

It is not difficult to show :

Fact: $B \in NP$ and $A \leq_{trans}^{poly} B$ implies $A \in NP$.

The importance of *NP*-completeness

To simplify the notation, lets sometimes use $\leq_{Cook}$ and $\leq_{Karp}$ respectively for genneral poly time

Basic Fact: If L is NP-complete (wrt to either $\leq_{Cook}$ or $\leq_{Karp}$, then $L \in P$ if and only if $P = NP$.

It can be shown that $L_{HC}, SAT, Partition, Vertex\text{-}Colour$ and thousands of other problems are NP complete.

So now we know that if we can polynomial time decide any one of these NP-complete problems we can solve them all in polynomial time. Even if a problem L is in NP but possibly not NP-complete, then $P = NP$ would imply L is polynomial time decidable.

But how do we prove that a decision propblem is NP-complete?

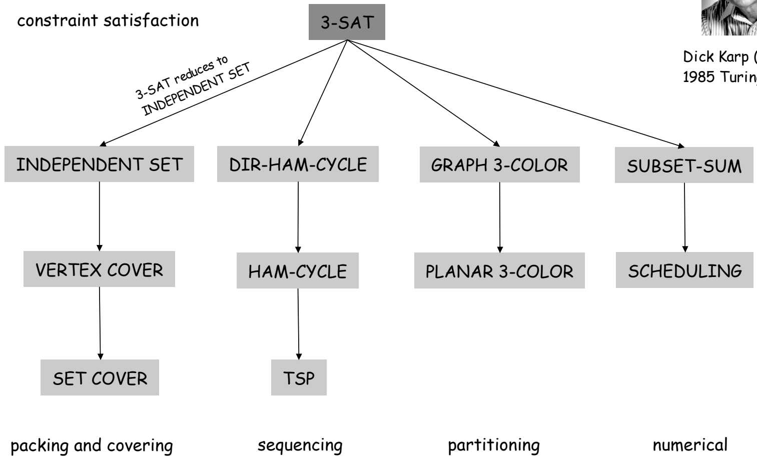
Suppose we know that some problem (for example, SAT) is NP-complete. Then if we can show SAT can be poly time reduced or transformed to (for example) $VC = \textit{vertex-cover}$, then VC must also be NP-complete.

Fact: Polytime reductions and polytime transformations are transitive relations. That is, for example, $A \leq_{Karp} B$ and $B \leq_{Karp} C$ implies $A \leq_{Karp} C$.

In this way, thousands of decision problems L have been created by a tree of polynomial time transformations. (On the next slide, we will show Karp's initial tree.) But we have to start the tree with some NP problem that we prove is NP-complete.

Cook did this for SAT by showing how to efficiently encode any polynomial time Turing machine computation of a verification predicate $R(x, y)$ and a "guess" for a certificate y within propositional logic. (We usually discuss this in more detail in CSC373.) Recall what we said about validity in predicate calculus (1st order logic).

A tree of reductions/transformations



Dick Karp (1972)
1985 Turing Award

End of Wednesday, November 16 class

We ended with the Karp tree of polynomial time transformations. Cook had established the NP-completeness of SAT and the transformation $SAT \leq_{T}^{poly} Clique$ which is basically the same as showing $SAT \leq_{T}^{poly} Independent\text{-}Set$.

I also “waved hands” at how we can show that *SAT* is *NP*-complete but you do not have to worry about that. The main idea of that result is to show how to encode polynomial time Turing machine computations by propositional formulas.

Next week we will complete our discussion of the *P* vs *NP* question and related issues. And then we will discuss complexity based cryptography.

Our final topic will be social networks. And after that we will just mention some other “great ideas” that we didn’t have time to discuss.

I am posting slides intended for the next two weeks. But I prefer to answer questions and not rush so we may not get to everything in the slides for week 10 and week 11. But these slides shoould help for the final assignment A4.